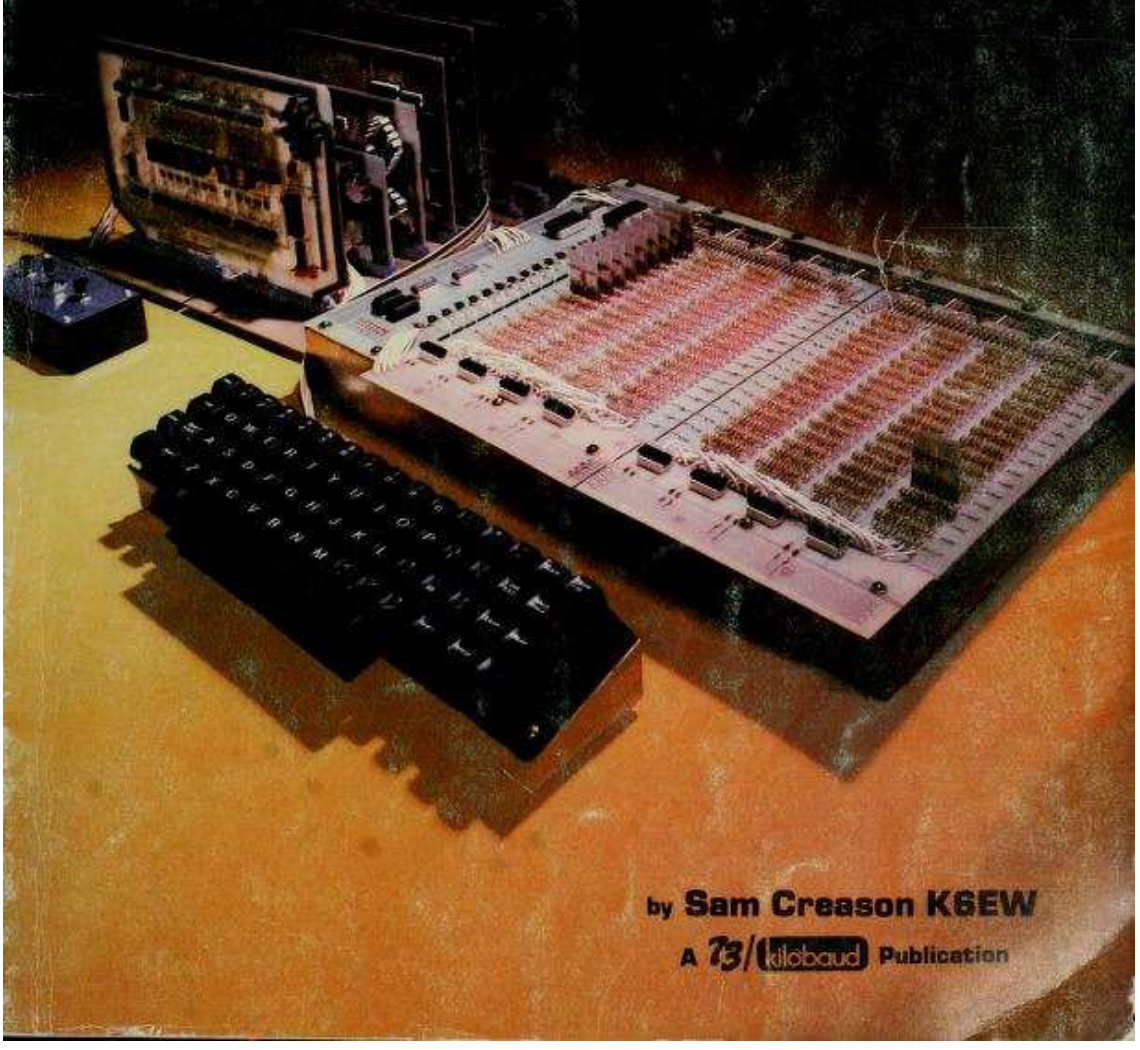
How to Build a Microcomputer

$9.95

. . . and Really Understand It



by Sam Creason K6EW

A 73/Kilobaud Publication

How to Build a Microcomputer

. . . and Really Understand It

by Sam Creason K6EW

For Pamela
A very special friend
who is also
A very special daughter

— Editor's Comments —

This book is unique. In this single publication, you'll find all the information and explanations you need to design, construct, test, program, debug, and operate your own microcomputer. Sam Creason's step-by-step approach means you'll not only be able to successfully duplicate the system shown on the front cover, but you'll also understand how it works when it's completed . . . a real plus.

—Circuit Boards Available—

The computer system described in this book was designed using only single-sided printed circuit boards to allow easy duplication with home etching systems. However, for those of you who prefer not to get involved in etching your own PC boards, we've arranged to make the boards available at very reasonable prices. For complete information on this, write to:

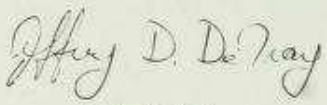
O. C. Stafford Electronics
427 South Benbow Road
Greensboro NC 27401

The Stafford Company is offering a special package price for an entire set of circuit boards, but they'll also sell individual boards. All of them are drilled, trimmed, and ready to go. If you want to save substantially and don't mind doing a bit of the work yourself, Stafford will also have undrilled boards available.

—Component Kits —

Finally, although Sam Creason's design uses readily available components, the Stafford Company can eliminate the problem of collecting them by providing a complete parts kit for this microcomputer. All parts are first quality, and the components for each board are packaged individually. Again, write to Stafford for the details.

There you have it. We've made it easy for you to enter the world of microcomputing. So turn the page . . . and get started!



Jeffrey D. DeTray
Editor

Contents

PREFACE

In my view, there is a gap in the information which is available to the novice builder of a small computer system. While considerable information is available concerning the generalities of how a computer works, how to program a computer and the like, details concerning the construction of a single, specific system are scarce. This book is intended to fill at least part of that gap. We'll proceed step by step through the implementation and application of an effective and easy-to-work-with microprocessor, the MOS Technology 6502. Aside from the 6502, only standard, inexpensive CMOS and TTL integrated circuits are used in the basic system. In that way, we can see and understand as much detail as possible.

Just as there is a gap in the information which is available to the novice, there is also a gap in the hardware which is used in the typical beginners' system. We'll see that the microprocessor is an extremely powerful tool once it's provided with instructions and other information. However, getting those instructions and the other information into the system so that we can use the power of the microprocessor can be a problem. With a set of toggle switches we can enter one word at a time, but the process is about as tedious as it is inexpensive. And that's particularly true if one of the instructions is incorrect so that the microprocessor erases the instructions instead of using them! Other possibilities include a keyboard and even a Teletype™. We can put information into the system more rapidly with either, but the possibility of accidental erasure still exists, and a Teletype™, at this time, costs about $900.

Our initial approach will be less conventional. We'll

build a diode-implemented, programmable, read-only memory. Using it, we can write programs and enter data by literally plugging words into memory. The result is very visible and easy to understand. Supplementing 64 to 256 words of such a memory with some conventional random-access memory and I/O devices (both digital and analog) produces a very efficient small system.

The system is useful in several applications in the amateur radio station. After the basic system is built, each subsequent addition of hardware involves a different one of these applications. Examples include a CW generator, a digital voltmeter, and a programmable signal generator. Of course, the system is useful for other purposes as well, and some of these are discussed in the final part of this book.

The information which follows is intended only as a description of a small computer system which this writer has built. Others have done similar work. The publications listed below are among those which contain descriptions of that work.

Kilobaud
73 Magazine
Electronics
EDN Magazine
Electronic Design
Digital Design
Mini-Micro Systems
Computer Design
Byte
Ham Radio
QST

Sam Creason K6EW

Chapter 1
Introduction

The Digital Computer

There are many ways to describe a digital computer. For our purposes, Fig. 1-1 is a useful representation. We describe the computer as a device which accepts numbers from the outside world via an input device — perhaps a keyboard, a set of switches, or even from memory as numbers which have previously been stored there. The computer performs some sort of predetermined manipulations in the microprocessor according to a set of instructions (a program) which is stored in memory. The computer then sends numbers back to the outside world via an output device — perhaps a Teletype printer, a TV display, or in another way which we'll find useful: a solid state switch. A control panel provides the means for an operator to supervise the system.

Fig. 1-1, in fact, is essentially a block diagram of our basic system.

Overview of Our Task

In the first part of the book we'll be concerned with the design and construction of the basic system. We'll also learn a little about programming it.

Since the language of the computer is numbers, our first order of business is to understand the forms that these numbers can take and the sort of manipulations which can be performed on and with them. Then, having learned a little about the numbers themselves, we'll study the properties of a few specific digital integrated circuits that will be used in our system and see how basic manipulations can be performed by using them. Because the expanded version of our system will contain linear as well as digital ICs, we'll also take time to examine the operational amplifier (op amp) and some of the circuits in which it's used.

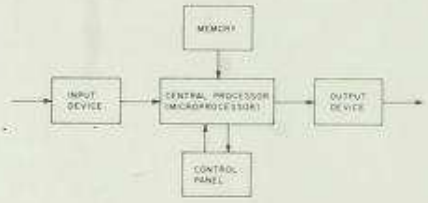


Fig. 1-1. Block diagram of a digital computer.

To begin putting our system together, we'll build the control panel and power supply. Having accomplished this, we'll build a rather unconventional memory — a diode-implemented, plug-in memory. It will allow us to quickly and easily communicate with the system before we add a keyboard or any other input device. We'll be able to plug words into memory as we please. This allows us to examine or change its contents easily, and avoids the problem of volatile data.

To reach the first plateau of our system, we'll add a third element — the MOS Technology 6502 microprocessor. We'll spend a little time discussing the 6502 and

then assemble it and one or two other ICs into a working configuration, completing our basic system.

We'll examine some basic instructions and see how they're combined into a working program. We'll write some very simple programs and watch as the computer performs them, one step at a time and then at full speed.

Improving our skills, adding more hardware, and programming the computer to do some useful tasks is the subject of the second part of the book.

We'll add some conventional random-access memory to our collection of hardware, and then write a program which will cause the computer to key the output of a CW transmitter with a predetermined message. We'll add a circuit which will accept a number from the computer and provide a related analog voltage as an output (a D/A — digital-to-analog converter). Using this circuit, we'll see how the computer will function as a signal generator. We'll add an A/D converter and digital display so that we can use the system as a digital voltmeter. We'll add a keyboard and program the system to function as a CW keyboard. Finally, in order to provide permanent storage for our programs, we'll add some conventional read-only memory.

In the third part of the book we'll consider other applications for which the system might be used, and how the system might be expanded.

Construction Techniques

Any of several methods of construction likely will produce a trouble-free system. Probably the simplest method is to use the printed-circuit layouts which are included in this book. None involve double-sided boards (even though they would be smaller) so that all can be made in the home workshop.

Alternatives include wire-wrapping, and using prototype boards which are available from the usual mail-order houses.

Two schools of thought are currently in vogue concerning the use of sockets for ICs. My own opinion is that if the builder plans to do much experimenting after the system is built, sockets should be used. An IC can't be ruined if it's temporarily unplugged from the system. Sockets with gold-plated contacts are preferable to those with tin-plated contacts. The latter tend to oxidize after a time and form high-resistance connections.

Testing The System

As each part of the system is built, we'll test it separately. In this way we can catch small problems before they become larger ones.

Some sort of logic probe or monitor is a necessity. An oscilloscope was useful in debugging the design, but should not be an absolute necessity if the circuits which are described in the book are used as is. A voltmeter is needed for calibrating one of the circuits in the expanded system.

ICs which are obtained from reputable mail-order houses are usually quite dependable. A good rule of thumb is to blame the circuit rather than the ICs, if a circuit doesn't work. Small cracks in the foil of a printed circuit board are particularly deceiving. I tried a half-dozen ICs in a particular circuit before it occurred to me to test the continuity of a foil run with an ohmmeter, for example.

Just as we should always suspect the circuit rather than the individual ICs, once that circuit has worked properly, we should blame the program rather than the circuit if the program produces unexpected results. Even seasoned programmers can't write error-free programs every time. In that regard, all programs for which machine code is shown in this book have been completely debugged. Then too, in the process of checking the proofs of the book, each code listing was taken from the proofs, entered into the system and the program was run, in order to avoid any possibility of typos.

In all projects of this nature, a source for at least one or two components is usually hard to find. Some helpful hints are included in Appendix II, including information on the availability of pre-fabricated, printed circuit boards.

Well then, that's what we'll build and how we'll build and test it. If we're careful and take our time, we'll have no serious problems. What's as important, we'll acquire the knowledge to read the hobby literature and be able to expand the system into a fairly sophisticated configuration.

Chapter 2
Basic Character Representations

The fundamental language of computers is made up of numbers. Before we can hope to use a computer or to understand how one works, we'll have to learn to speak the language of binary numbers and some of its dialects – hexadecimal, binary-coded decimal, and ASCII. That's our objective in this chapter.

The Decimal System

Although we want to learn to use binary numbers, let's first briefly examine the decimal system. In doing so, we'll learn about the structure of a number system and be able to understand the binary system with less effort.

The decimal system uses ten digits: 0 through 9. Even with this limited set, we can write numbers (groups of digits) which refer to quantities that are as large as we please. We do this by using a positional notation. That is, while each digit has a basic meaning associated with it, that meaning is modified by the position which it occupies in the number. For example, each 5 in the number 5555 has the same meaning when taken alone. However, if we consider the number as a whole, then each five has a different meaning. As we move from right to left in the number, each five is understood to be multiplied by a successively higher power of ten. That is:

$$5555 = 5\times10^3 + 5\times10^2 + 5\times10^1 + 5\times10^0$$

Since the left-most digit is multiplied by the highest power of ten, we call it the most significant digit. Similarly, the right-most digit is called the least significant digit. Since successive powers of ten are used as multipliers, we say that the decimal system has a base of ten.

The Binary System

While the decimal system seems "natural" to us, that's really only because it's the first system which we learned to use. In principle, any number can be used as the base of a number system. The binary system uses 2 as a base. By analogy to the decimal system, only two digits are needed in the binary system: 0 and 1. This is a particularly convenient system to use in a computer, since many devices are two-state in nature. A switch is on or off, for example.

We determine the value of a binary number in the same way that we determine the value of a decimal number. However, the multipliers are powers of 2 rather than powers of 10. For example:

$$1010 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$$

When there is a possibility of confusion, we'll specify the base which we're using by a subscript (decimal value) to the right of the number. For example:

$$1011_2 = 11_{10}$$

Binary-Decimal and Decimal-Binary Conversion

We'll occasionally want to convert decimal numbers to binary and vice versa. Of the two processes, binary-to-decimal conversion is the easier. We simply make use of the definition of a binary number. For example:

$$1010_2 = 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 =$$
$$8_{10} + 0_{10} + 2_{10} + 0_{10} = 10_{10}$$

Any binary number can be converted to the cor-

responding decimal number by the same process.

Decimal-to-binary conversion isn't much harder. All we need is a table of the decimal equivalents of the powers of 2, as in Fig. 2-1. We subtract from the decimal number the highest power of 2 that will result in a positive (or zero) remainder, make a note of the power of 2, and repeat the process on the successive remainders until there is none. We then construct the binary number. For example, let's convert 78 to its binary equivalent. The highest power of 2 which we can subtract without causing a negative remainder is 2^6, or 64. The remainder is 14. We subtract 2^3 (8) from 14, leaving 6. We subtract 2^2 (4), leaving 2. We subtract 2^1 (2), leaving 0. Thus:

$$78_{10} = 1x2^6 + 0x10^5 + 0x10^4 + 1x2^3 + 1x2^2 + 1x2^1 + 0x2^0 = 1001110_2$$

Arithmetic With Binary Numbers

As we write computer programs, we'll occasionally need to do binary arithmetic. Since there are only two digits in the binary number system, it's quite simple.

In binary addition, there are only three combinations:

```
          1
  0   0   1
 +0  +1  +1
 --  --  --
  0   1   0   (carry 1)
```

To add multi-digit numbers, we proceed digit-by-digit, making carries as necessary, just as in decimal addition. For example:

```
    1
  100 | 1
 +010 | 1
 ------------
  111 | 0
```

Similarly:

```
  1   1   1   1   1
+ 1   1   1   1   1
 ---------------------
  1 | 1 | 1 | 1 | 0
```

So much for addition.

We can perform binary subtraction in more than one way. The method we'll use makes use of the fact that subtracting one positive number from a second is equivalent to adding the negative of the first to the second number. Subtraction thereby is reduced to addition, and the same circuit within a computer can perform both operations.

The method is called the method of complements. To understand it, let's first examine the method using decimal arithmetic. To do this, we first have to consider how negative numbers can be represented. Let's restrict ourselves to, say, four digits. If we start counting at 0000, we'll eventually reach 9999, and then 0000 again. Let's use the first half of our set of numbers as positive, and the second half as negative. Since 9999 comes just before 0000, it's the smallest negative number, as shown in Fig. 2-2. To subtract 1 from 3, for example, we add 3 and the equivalent of -1:

```
  1   1   1   1
  0 | 0 | 0 | 3 |  3
  9 | 9 | 9 | 9 | -1
 ---------------------
  0 | 0 | 0 | 2 |  2
```

We ignore the final carry, since we've restricted ourselves to no larger than four-digit numbers.

A similar table of binary numbers is shown in Fig. 2-3. Note that the most significant digit of a negative number is 1 and the most significant digit of a positive number is 0, if we use this convention.

To subtract 0001 from 0011, we add 0011 and the equivalent of -0001:

```
  1   1   1   1
  0 | 0 | 1 | 1
  1 | 1 | 1 | 1
 ---------------
  0 | 0 | 1 | 0
```

Again, we ignore the final carry.

The negative equivalent in the binary system is called the two's complement. We obtain it by first setting each digit of the number to the opposite value, which produces the one's complement. We then increment (add 1 to) the one's complement.

```
complement
00101      11010

increment
   11010
  +00001
  ------
   11011
```

Thus, -00101 and 11011 represent the same value.

Binary multiplication and division are per-

```
2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
2^10 = 1024
2^11 = 2048
2^12 = 4096
2^13 = 8192
2^14 = 16384
2^15 = 32768
2^16 = 65536
```

Fig. 2-1. Decimal equivalents of the powers of two.

```
4999
 |
 |
 |
 |
 |
0003
0002
0001
0000
9999
9998
9997
 |
 |
 |
 |
 |
5000
```

Fig. 2-2. Complementary notation; decimal numbers.

```
0111
 |
 |
 |
 |
 |
0011
0010
0001
0000
1111
1110
1101
 |
 |
 |
 |
 |
1000
```

Fig. 2-3. Complementary notation; binary numbers.

formed in ways which are analogous to those used in decimal arithmetic. In general, we'll not need to know how to do either. Except for a special case, we'll not even consider them further. The special case concerns multiplication or division by an integral power of 2.

We recall that in decimal arithmetic, shifting each digit of a number to the left one place corresponds to multiplying the number by 10. Two such shifts correspond to multiplication by 100, and so on. Similarly, shifting each digit to the right one place corresponds to dividing the number by 10. In the binary system, shifting the digits corresponds to multiplying or dividing the number by a power of 2. Thus:

corresponds to dividing 010100_2 by 2^2, since each digit is shifted two places. Trailing 0s are dropped and leading 0s are supplied. Similarly:

corresponds to division by 4. In this case, since the number is negative, leading 1s are supplied, so as to avoid converting it to a positive number. Note that if the least significant digit of the original number is 1, the number is odd and can't be divided by any integral power of 2.

We perform binary multiplication by an integral power of 2 by shifting each digit to the left. Thus:

corresponds to multiplying 001100_2 by 2^1, since each digit is shifted once. A trailing 0 is supplied and the leading 0 is dropped. We have to be careful when performing multiplication. The most significant digit of the result must be the same as the most significant digit of the original number or we've done something other than multiply the number by an integral power of 2. For example:

$$\begin{array}{ll} 1010 & 10_{10} \\ 0100 & 4_{10} \end{array}$$

$$\begin{array}{ll} 0110 & 6_{10} \\ 1100 & 12_{10} \end{array}$$

In the second case, the result is correct provided we're not working with two's complements, in which case the most significant figure must be a "0".

Logical Operations With Binary Numbers

It's possible for us to get by without knowing a lot about binary arithmetic. However, there are other operations on binary numbers which we must thoroughly understand if we're to understand anything at all about our system. These operations are called *logical operations*.

The simplest logical operation which we can perform is the NOT or complement operation. To perform it, we simply set each digit of a binary number to its opposite value. The NOT operation is signified by placing a bar over the number which is to be operated on. That is:

$$\overline{0} = 1 \text{ and } \overline{1} = 0$$

If we consider more than just a single binary number, other operations are possible. The first which we'll examine is the OR operation. If we OR two one-digit numbers, the result is defined as 1 if either one digit or the other (or both) is 1. Otherwise, the result is defined as 0.

The operation can involve as many digits as we like. For example, the results for the possible combinations of three variables is shown in Fig. 2-4.

Such a table is called a truth table. It can be summarized by writing:

$$R = A + B + C$$

The "+" signifies the OR operation.

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Fig. 2-4. Truth table for $R = A+B+C$.

A	B	R
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 2-5. Truth table for $R = \overline{A+B}$.

A	B	R
0	0	0
0	1	1
1	0	1
1	1	0

Fig. 2-6. Truth table for $R = A \oplus B$.

A related logical operation is the NOR operation, where NOR is a contraction of NOT and OR. The NOR operation involves ORing the digits and NOTing the result. The truth table for two variables is shown in Fig. 2-5. The result is 0 if either A or B is 1, otherwise the result is 1. That is:

$$R = \overline{A + B}$$

When we discuss the design of the system, we'll come across an IC called a "two-input NOR gate." It's simply a device which produces a 1 at the output if both inputs are 0.

A related logical operation is the EXCLUSIVE — OR operation, abbreviated EOR. The truth table for two variables is shown in Fig. 2-6. The result is 1 only if one of the inputs is 1. If both are 1s (or 0s), the result is 0. That is:

$$R = A \oplus B$$

Where the "$\oplus$" signifies the EOR operation. We'll use the EOR operation only infrequently.

A logical operation which we'll use quite fre-

A	B	R
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 2-7. Truth table for $R = A \cdot B$.

quently is the AND operation. The truth table for two variables is shown in Fig. 2-7. The result is 1 only if both A and B are 1. That is:

$$R = A \cdot B$$

Where the "·" signifies the AND operation.

We'll also encounter the NAND operation. As the name implies, it's a combination of the AND and NOT operations. The truth table for two variables is shown in Fig. 2-8. That is, the result is 0 if both A and B are 1, otherwise the result is 1.

A	B	R
0	0	1
0	1	1
1	0	1
1	1	0

Fig. 2-8. Truth table for $R = \overline{A \cdot B}$.

Till now, we've applied the logical operations to a single digit or groups of digits. They also may be applied to multi-digit numbers and groups of such numbers. The operation simply is carried out digit by digit or on corresponding pairs of digits:

$$\overline{1010} = 0101$$
$$1010 + 1100 = 1110$$
$$1010 \circ 1100 = 0110$$
$$1010 \cdot 1100 = 1000$$

The Hexadecimal System

At this point we may begin to feel that while binary numbers may be useful to a computer, they can be quite cumbersome to humans. Our computer will use eight- and sixteen-digit (one- and two-byte) binary numbers. While we might be able to keep track of eight 1s and 0s at a time, we'd have problems with sixteen.

The solution is to use a number system which has more than just two digits in its set of symbols. If we use a power of 2 as the base, then translation to and from binary numbers will be quite simple. The hexadecimal system is such a system. As its name implies, the number sixteen is used as a base. Since the base is 16, the set of symbols must include sixteen different digits. For the first ten, the digits zero through nine are used. For the remaining six, the first six letters of the alphabet are used. The first eighteen hexadecimal numbers are shown in Fig. 2-9, along with the corresponding binary and decimal numbers for comparison.

We could start from scratch and develop the properties of the hexadecimal system in the same way that we developed the properties of the binary system, but that's not necessary. Rather, we can discuss the hexadecimal system in terms of its relationship to the binary system.

To write the hexadecimal equivalent of a binary number, we divide the binary number into sets of four digits, starting with the least significant digit. We supply leading digits as necessary, so that the left-most set contains four digits. We then write the hexadecimal digits which correspond to each group of four binary digits. These make up the hexadecimal number. For example:

binary number:
01101010011101
groups of four digits:
0001 1010 1001 1101
hexadecimal number:
1 A 9 D

We'll make continuous use of hexadecimal numbers as we write programs for our computer.

DECIMAL	BINARY	HEXADECIMAL
0	00000	0
1	00001	1
2	00010	2
3	00011	3
4	00100	4
5	00101	5
6	00110	6
7	00111	7
8	01000	8
9	01001	9
10	01010	A
11	01011	B
12	01100	C
13	01101	D
14	01110	E
15	01111	F
16	10000	10
17	10001	11

Fig. 2-9. First eighteen numbers in three systems.

Binary-Coded Decimal (BCD) Numbers

Even though our computer won't be able to use decimal numbers, there will be times when we'll want an output to

DECIMAL	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Fig. 2-10. The BCD numbers.

bear some similarity to a decimal number. For this purpose, we'll use the *natural binary decimal code*. The ten decimal digits simply are represented by their corresponding four-digit binary numbers, as shown in Fig. 2-10.

To represent a multi-digit decimal number, we write the four-digit binary equivalent of each decimal digit in turn. For example:

decimal number:
7 4 8
BCD equivalent:
0111 0100 1000

BCD is a somewhat inefficient means by which to represent numbers. Four binary digits can represent only ten different numbers in BCD, while the same four digits can represent sixteen different binary numbers.

The ASCII Code

The final topic which we'll consider in this chapter is the ASCII code. It's the means by which we'll be able to exchange alphanumeric information with our system.

For example, the letter M isn't included in any of

the number systems which we've discussed. Yet if we write a program which accepts input from a keyboard, we'll need sets of 1s and 0s to represent not only M, but the other letters of the alphabet, the decimal digits, and punctuation marks as well.

A code which is called the American Standard Code for Information Interchange, abbreviated ASCII, was devised for this purpose. It consists of 128 seven-bit binary numbers, each of which represents an alphanumeric symbol. The portion of the code which will be useful to us is shown in Fig. 2-11.

CHARACTER	ASCII	CHARACTER	ASCII
Line Feed	0A	@	40
Carriage Return	0D	A	41
Space	20	B	42
!	21	C	43
"	22	D	44
#	23	E	45
$	24	F	46
%	25	G	47
&	26	H	48
'	27	I	49
(	28	J	4A
)	29	K	4B
*	2A	L	4C
+	2B	M	4D
,	2C	N	4E
-	2D	O	4F
.	2E	P	50
/	2F	Q	51
1	31	R	52
2	32	S	53
3	33	T	54
4	34	U	55
5	35	V	56
6	36	W	57
7	37	X	58
8	38	Y	59
9	39	Z	5A
:	3A	[	5B
;	3B	\	5C
<	3C	]	5D
=	3D	↑	5E
>	3E	←	5F
?	3F		

Fig. 2-11. Hexadecimal representation of a portion of the ASCII code.

Chapter 3
Basic Hardware

In the previous chapter, we learned a little about binary numbers and some of the operations which can be performed on them. In this chapter, we'll examine some simple digital devices which perform these operations and discuss the properties of the two families of devices which we'll use in our system.

Then, since we'll use op amps in the expanded version of our system, we'll discuss them and some of the circuits in which they are used.

Finally, we'll catalog the functions and pin designations of the small-scale digital devices and op amps of interest.

Digital Logic Elements

Just as the simplest logical operation is the NOT operation, the simplest logic element is the one which performs that operation. It's called a NOT gate, or inverter. The output of the inverter is always the complement of its input. We symbolize it as in Fig. 3-1.

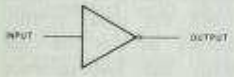


Fig. 3-1. The inverter.

Actually, there is a simpler example of a logic element than an inverter. It's similar, except that its output is always in the same state as its input. The buffer is an example of such a device. It's not used to implement a logical function, rather it's used to drive a heavier load then other elements can. We symbolize it as in Fig. 3-2.

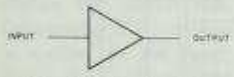


Fig. 3-2. The buffer.

The inverter and the buffer each operate on a single digit – a single bit, in the language of hardware. They're the only useful devices with a single input.

If we consider gates with two or more inputs, we can implement all the logical operations which were discussed in Chapter 2. Two-, four-, and eight-input OR, NOR, AND and NAND gates, and two-input EOR gates are commonly available. The two-input versions are symbolized as in Fig. 3-3. Symbols for the four- and eight-input versions are similar.

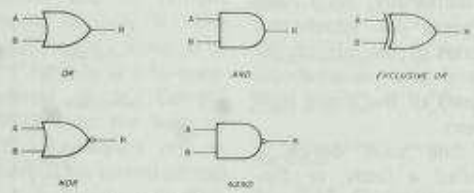


Fig. 3-3. Two-input logic gates.

Using inverters and gates, we can implement a wide variety of more complex functions. One such is shown in Fig. 3-4. This particular circuit, called a decoder, accepts a two-bit binary number as an input and provides a series of unique single-bit outputs as summa-

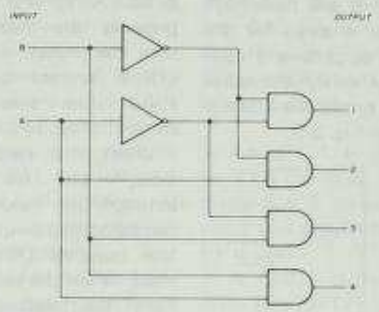


Fig. 3-4. A decoder.

rized in the truth table in Fig. 3-5.

INPUTS		OUTPUTS
A	B	1 2 3 4
0	0	1 0 0 0
0	1	0 1 0 0
1	0	0 0 1 0
1	1	0 0 0 1

Fig. 3-5. Truth table for decoder.

We can verify the table by determining the states of the inputs of each AND gate as a function of the two-bit input to the decoder circuit. For example, output 4 will be high only if both inputs A and B are high. No other output will simultaneously be high, since at least one input to each of the other AND gates will be low when inputs A and B are both high. Similar analysis for the other outputs will completely verify the table.

We symbolize a decoder as in Fig. 3-6.

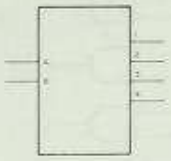


Fig. 3-6. Symbol for a decoder.

In our system, we'll use a number of decoders, each contained in a single IC. They'll be more complex than our example, though, decoding four inputs to produce sixteen outputs.

One of the characteristics of our system is that the various modules communicate via a shared circuit. It's used in much the same way as a telephone party-line. That is, only one device may transmit on the circuit at any given time. The remaining devices which are capable of transmitting must be electrically disconnected from the circuit at that time. To accomplish this, we connect each device (which transmits) to the circuit via a solid state switch.

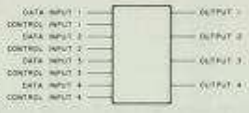


Fig. 3-7. Symbol for a solid state switch.

Typically, these are four-bit devices (four inputs and corresponding outputs). A control input provides the means to turn the switch on and off. A symbol for the solid state switch is shown in Fig. 3-7.

When the switch is "on," the resistance between an input and corresponding output is a few hundred Ohms, at most. When the switch is "off," the resistance is at least several hundred megohms.

For the devices which we've discussed, the states (logic levels) of the outputs at any time depend on the states of the inputs at that same time. Additionally, we'll need devices for which the states of the outputs depend also on what the states of the inputs have been.

One such device is called a latch, or flip-flop. It has one or more data inputs and a corresponding number of data outputs. A control (or clock) input provides the link between a data input and the corresponding output. Data is transferred from an input to the output when an appropriate signal is applied to the control input. Once the state of the output has been "updated" in that way, it can't change again until another signal is applied to the control input, regardless of what happens at the data input.

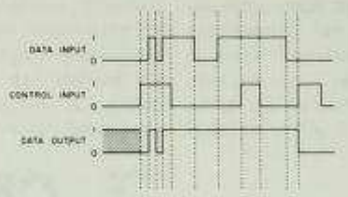


Fig. 3-8. Timing diagram for latch.

One such device is called a level-sensitive latch. When the control input is taken to one logic level, the state of the output is the same as the state of the input. The output tracks the input. However, when the control input is taken to the other logic level, the output retains the value that it has when the transition occurs at the control input. The behavior of a latch with an active-high control input is shown in Fig. 3-8. A symbol for the device is shown in Fig. 3-9.

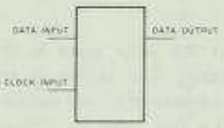


Fig. 3-9. Symbol for a latch.

Another storage element which we'll use is a type D (data) flip-flop. It's similar to the latch which we just discussed, but it's actuated in a slightly different way. It transfers data from an input to the corresponding output in response to a transition from one logic level to the other on the control input. The output can't be made to track the input continuously as was the case with the level-sensitive latch.

The behavior of a positive-edge triggered flip-flop is shown in Fig. 3-10. We use the same symbol as for the level-sensitive latch, since the

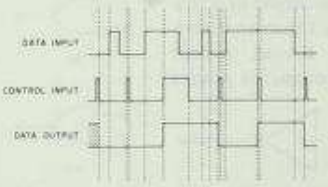


Fig. 3-10. Timing diagram for flip-flop.

type-number of the device indicates which sort it is.

These, then, are the small-scale digital devices which we'll use throughout the system. We'll also use a few specialized large scale devices on particular boards, but we'll postpone discussing them until we examine the particular circuits in which they're used.

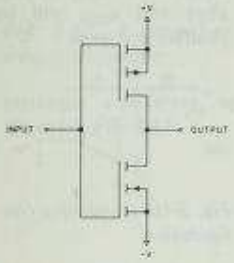


Fig. 3-11. A CMOS inverter.

Families of Logic Elements

The large majority of devices which we'll use in our system belong to the CMOS family. As the name implies, they are implemented with MOS transistors. The C stands for complementary. In an inverter, for example, a p-channel MOS transistor and an n-channel MOS transistor — a complementary pair — are combined in one circuit, as shown in Fig. 3-11.

If a high level is applied to the input, the upper transistor is turned off and the lower transistor is turned on. This clamps the output low. Conversely, if a low level is applied to the input, the output is clamped high.

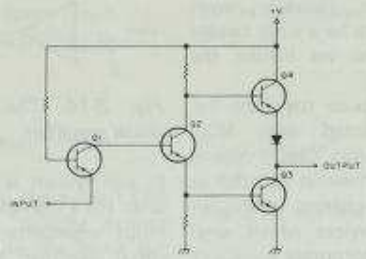


Fig. 3-12. A TTL inverter.

Since MOS transistors are voltage operated, the circuit which drives such a device needs supply only enough current to charge or discharge the input capacitance of the transistors. When the transistors aren't being switched, essentially no current flows in the input circuit. Under these conditions, power dissipation is measured in fractions of a microwatt per gate. Because of this, most CMOS devices aren't designed to source or sink currents of more than a mA or so.

In one application, though, we'll need devices which can tolerate higher currents. For that purpose, we'll use members of the TTL (Transistor-Transistor Logic) family. The circuit of a TTL inverter is shown in Fig. 3-12.

If the input is taken high, the base-emitter junction of Q1 is reverse biased and no current flows through it. The base-collector junction of Q1 behaves as a forward-biased diode. Current flows into the base of Q2, turning it on. In turn, Q3 is turned on and Q4 is turned off. This clamps the output to within a few tenths of a volt of ground. A similar analysis will indicate that if the input is taken low, the output will be high. High in this case means about 3.3 volts, since there are several diodes in the path between the +5 volt terminal and the output terminal.

In contrast to MOS transistors, the bipolar transistors which are used in TTL devices are current operated. A steady state current of 1.6 mA is required in order to hold the input of a medium power TTL gate low, and power dissipation is measured in tens of milliwatts per gate.

If we use devices from just one family, we don't have to be overly concerned with the characteristics of the family. However, if we intermix CMOS and TTL devices, we have to take differences into account. Some pertinent data are summarized in Fig. 3-13.

Two types of devices are shown for each family. Most of the CMOS devices which we'll use fall into the category of "gate," but we'll use buffers for interfaces to the real world and to TTL devices.

The TTL devices which we'll use fall into the category of low power TTL(LPTTL). The characteristics of medium power devices are shown for comparison, because when we speak of TTL, we usually mean medium power TTL(MPTTL).

From the characteristics which are shown in Fig. 3-13, we can make the following conservative generalizations about mixing families:

1. An LPTTL or a regular TTL device will drive many CMOS inputs provided that a pull-up resistor is used.
2. A CMOS buffer will drive two MPTTL inputs or a dozen or so LPTTL inputs.
3. A CMOS gate will drive an LPTTL input. It

	CMOS		TTL	
DEVICE	GATE	BUFFER	MEDIUM POWER	LOW POWER
input hi voltage	>3.5	>3.5	>2.0	>2.0
input lo voltage	<1.5	<1.5	<0.8	<0.8
input hi current	<1 μA	<1 μA	40 μA	4 μA
input lo current	<1 μA	<1 μA	1.6 mA	0.18 mA
output hi voltage	4.99	2.5*	3.3	3.3
output lo voltage	0.01	0.4*	0.2	0.2
output hi current	<1 mA	2.6 mA	400 μA	40 μA
output lo current	<1 mA	6 mA	16 mA	3.6 mA

*As load decreases, these values approach those which are shown for the CMOS gate.

Fig. 3-13. Characteristics of CMOS and TTL devices. Supply voltages are +5 Vdc.

will not drive an MPTTL input.

As we mentioned earlier, CMOS devices consume very little power. They also have other properties which make them attractive for our system. For example, they're rather slow compared to MPTTL devices (but not that much slower than LPTTL devices). While that may seem like a limitation, it's a useful characteristic to us. Because of it, we can be quite casual about the lengths and types of connections which we make between devices. Flip-flops can't be falsely triggered by high frequency noise because they can't react quickly enough. Reflections which are caused by transmission line effects are simply absorbed during transitions because the latter require a relatively long time to complete.

As a practical rule of thumb, CMOS devices can be operated at frequencies no higher than about 2 MHz if +5 volt power is used. At that frequency we can use conductors up to about three feet long before we have problems.

CMOS ICs also are fairly insensitive to a poorly-regulated power supply. However, other devices in our system do require well-regulated power. Since supplying well-regulated power to all devices is no more difficult than supplying it to just a few, we'll take that course. Because of all this, CMOS devices are the logical choice for our system. However, we'll have to be a little careful in how we handle the ICs.

Because they are implemented with MOS transistors, CMOS devices are somewhat sensitive to static charges. While all the devices which we'll use incorporate some sort of internal protection against static discharge, it's well to treat them with respect. For example, all CMOS devices which I've ever seen were placed in a conductive tube, wrapped in foil, or inserted in conductive foam before shipment. It's extremely good practice to provide such protection whenever the ICs are not in place in a circuit. Of course, even if an IC is in place on a PC board, it may not be adequately protected if the PC board itself is not inserted in its own socket.

A ground lead should be clipped to the tip of a soldering iron before the iron is used if any CMOS devices are in place in the circuit.

If we're reasonably careful, we'll not have problems. For example, during the design and construction of the prototype, only one CMOS IC was ruined, due to reversed power leads.

At this point, we have in hand most of what we need to know about digital ICs. Let's turn now to a discussion of linear ICs.

The Operational Amplifier

The linear device which we'll make use of

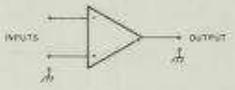


Fig. 3-14. The operational amplifier.

in our system is the op amp. It's a high-gain, high input impedance, differential amplifier. We represent it as a three terminal device (plus power connections) as shown in Fig. 3-14.

The input terminal which bears a negative sign is called the inverting input. The other is called the non-inverting input. If we ground the non-inverting input and apply a negative voltage to the inverting input, the output will swing positive. Conversely, the output will swing negative in response to a positive voltage at the inverting input.

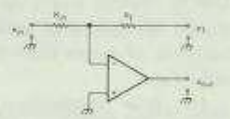


Fig. 3-15. Development of a working configuration.

In Fig. 3-15 the inverting and non-inverting inputs are connected to a voltage divider and ground, respectively. The potential of the inverting input, the potential difference of the two inputs, and the value of e_{out} are thus determined by the relative values of e_{in} and e_1, and R_{in} and R_f. In particular, the inverting input is at ground potential provided that:

$$e_{in}/R_{in} = -e_1/R_f$$

as application of Ohm's Law to the voltage divider will show. If we make e_{in}/R_{in} slightly more positive than $-e_1/R_f$, then the inverting input will be slightly positive with respect to ground and e_{out} will be negative with respect to ground and relatively large. Similarly, if we make e_{in}/R_{in} slightly more negative than $-e_1/R_f$, then the inverting input will be slightly negative and e_{out} will be positive and large.

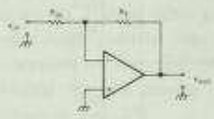


Fig. 3-16. A working configuration.

In Fig. 3-16, the output of the amplifier is connected to one end of the voltage divider. If e_{in} is initially zero, e_{out} is zero. If we make e_{in} positive, the inverting input will become positive and e_{out} will become negative. But if e_{out} becomes too negative, the inverting input will become negative and e_{out} will become positive. As we change e_{in}, e_{out} changes in response, always taking on a value which will drive the potential of the inverting input to zero. Thus, the working equation for the design of op amp circuits in which the non-inverting output is grounded is:

$$e_{out} = (-R_f/R_{in})\, e_{in}$$

By Ohm's Law:

$$i_{in} = -i_f$$

That is, the current in the feedback loop is equal in

magnitude and opposite in sign to the current in the input circuit.

By proper choice of feedback and input elements, we can design circuits which will perform a number of functions. For example, we can implement a current-to-voltage converter as shown in Fig. 3-17.

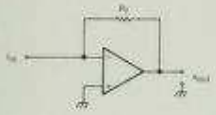


Fig. 3-17. A current-to-voltage converter.

In this case:

$$-e_{out}/R_f = i_{in}$$

Or

$$e_{out} = R_f(i_{in})$$

That is, the output voltage is directly proportional to the input current.

If the non-inverting input of an op amp is at other than ground potential, e_{out} will assume a value such that the inverting input is driven to that same potential. Using that property, we can implement an analog buffer as shown in Fig. 3-18.

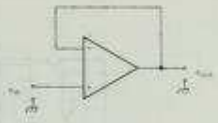


Fig. 3-18. An analog buffer.

In this case:

$$e_{out} = e_{in}$$

The circuit of an analog comparator is shown in Fig. 3-19. In this case the op amp is used in an open loop configuration. Since the non-inverting input is tied to ground potential, the state of the output indicates whether the applied voltage is positive or negative.

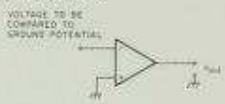
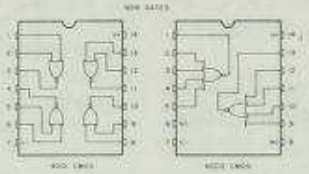


Fig. 3-19. A comparator.

With that, we've examined the small-scale building blocks which we'll use in our systems. The remainder of the chapter contains abbreviated spec. sheets on the particular devices which we'll use.

ALL DEVICES VIEWED FROM TOP EXCEPT AS NOTED



Each individual gate may be used independently. For each, the output is low if any input is high.

Each individual gate may be used independently. For each, the output is low if all the inputs are high.

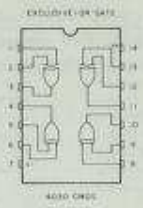
The package contains two independent devices. Each is a positive-edge-triggered type D flip-flop.

Data which is present at the D input when a low-level to high-level transition is applied to the CLOCK input is stored and appears at the Q output an instant later. The output cannot be made to track the input continuously. The $\overline{Q}$ output provides the complement of Q.

Holding the SET input high holds the Q output high ($\overline{Q}$ low) unconditionally. Holding the RESET input high holds the Q output low ($\overline{Q}$ high).

Each switch may be used independently. For each, the resistance between corresponding input and output terminals is about 300 ohms if a high level is applied to the corresponding CONTROL input. If a low level is

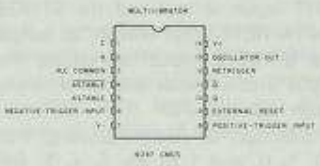
applied to the CONTROL input, the resistance is several thousand megohms.



Each individual gate may be used independently. For each, the output is high if either (but not both) input is high.

The package contains four elements, each of which has an independent data terminal. All four elements share a common CLOCK input. Q and $\overline{Q}$ outputs are available.

The device is programmed to be high-level or low-level triggered depending on the level to which the POLARITY input is strapped. If that level is high, the Q outputs follow the corresponding D inputs when the CLOCK input is taken high, for example. If the CLOCK input is then taken low, the existing values at the D inputs are stored.

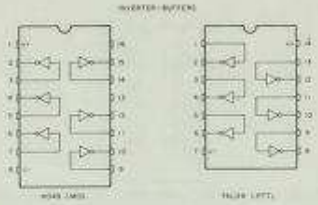


The device is a gated astable multivibrator. Internal logic permits its use as a monostable multivibrator as well.

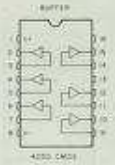
For use as a positive-edge-triggered monostable multivibrator, pins 4 and 14 are tied to V- and pins 5, 6, 7, 9, and 12 are tied to V+. The trigger pulse is applied to pin 8. For use as a negative-edge-triggered monostable multivibrator, pins 4, 8, and 14 are tied to V+ and pins 5, 7, 9, and 12 are tied to V-. The trigger pulse is applied to pin 6. In either case the output may be taken from pin 10 or 11.

The duration of the output pulse in seconds is

approximately 2.5 RC, where R is the value of the timing resistor in megohms and C is the the value of the timing capacitor in microfarads.



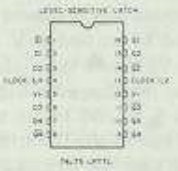
Each gate may be used individually. For each, the output is high if the input is low, and vice-versa.



Each gate may be used separately. For each, the output is high if the input is high, and vice-versa.

The device accepts a four-bit binary input. One output goes high in response to the input, provided that the INHIBIT input is low and the STROBE input is high. If the INHIBIT input is high, all outputs are low. Taking the STROBE input low stores the currently applied input into internal latches until the STROBE input is taken high again.

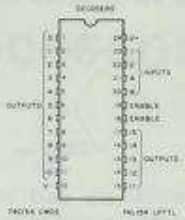
Output selection is straightforward. For example, applying 0, 1, 0, and 1 to the 8, 4, 2, and 1 inputs, respectively, selects output number 5.



The package contains four elements. Each has an

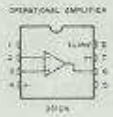
independent set of data terminals: D input, and Q and $\overline{Q}$ outputs. Pairs of elements share common CLOCK inputs.

If a CLOCK input is high, the outputs of the corresponding elements follow their respective inputs. If a CLOCK input is taken low, the current values of inputs are stored until that CLOCK input is taken high again.



The device accepts a four-bit binary input. One output goes low in response to the input, provided both ENABLE inputs are low. If either ENABLE input is taken high, all inputs are high, unconditionally. If one ENABLE is taken low, the selected output will follow the data which is applied to the other ENABLE input.

Output selection is straightforward. For example, applying 0, 1, 0, and 1 to the 8, 4, 2, and 1 inputs, respectively, selects output number 5.

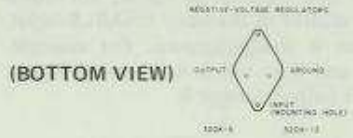


The device is a general-purpose operational amplifier. Large-signal frequency response is flat to about 80 kHz. The output may be clamped within any desired limits to make it compatible with logic circuits. An external circuit to provide dc balance of offsets may be used but is not required.

The device is a general-purpose operational amplifier. Large-signal frequency response is flat to about 10 kHz. An external circuit to provide dc balance of offsets may be used but is not required.

The 2N2222 is an NPN transistor which is rated for service at collector-to-emitter voltages up to 40 volts, and dissipations up to 1.8 watts at 25° C. Current gain is 30 (minimum) at a frequency of 1 kHz when the collector current is 1 mA. Switching times are less than 100 nanoseconds.

Two different devices are used in our system to produce two different regulated voltages: +5 and +12 volts. Each is a 3-terminal device, with pinouts as shown above. Each can pass currents up to about one Amp, if a heatsink is used. These devices may run quite warm.



Two different devices are used in our system to produce two different regulated voltages: -5 and -12 volts. Each is a 3-terminal device, with pinouts as shown above. Each can pass currents up to about one Amp, if a heatsink is used. These devices may run quite warm.

Chapter 4
Introduction to Microprocessor Systems

At this point, we have in hand much of the basic information which we'll need. We could understand how the circuit boards which we'll build provide their respective functions. However, we'd probably not understand why the various functions are necessary. It's appropriate, then, to examine the basics of a microprocessor system in more detail than was provided in Chapter 1.

Building Blocks

A block diagram of a basic system is shown in Fig. 4-1.

The microprocessor (uP) may be described as the "brain" of the system. It's fundamentally nothing more than a collection of gates, flip-flops, and other logic elements, all fabricated together on one chip. The function of the uP is to carry out a program or set of instructions, (in the form of a list of numbers) which we provide for it. For example, a simple program might cause the uP to fetch two numbers from the outside world, add them, and return the result to the outside world.

The uP contains an oscillator, or clock, so that all this can be done in an orderly, timed and synchronized sequence.

In a few words, that's what uP does. How it does it is the material for another book.

The block in the basic system which holds the program for the uP is the *read-only memory* (ROM). It may also hold other numbers which will be operated on. It makes available instructions or other numbers, one at a time, on command from the uP.

The next block in the system is the *input port*. The circuit is made up of latches and solid state switches. The input terminals of the latches are available to us so that we can write a number into the latches. The outputs of the latches are available to the uP via the solid state switches. The uP can read the number by simply turning on the switches.

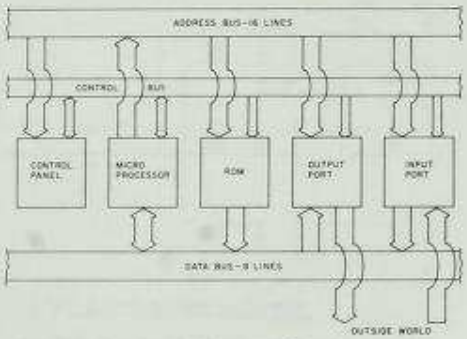


Fig. 4-1. Block diagram of basic system.

Numbers are gotten out of the system via the next block, the *output port*. The circuit is made up of latches into which the uP can write a number. The output terminals of the latches are available to us.

The final block in our basic system is the control panel, It exchanges control signals with the uP. Through it, we can supervise the operation of the uP.

Paths for Communication

The various blocks communicate with each other via three paths, called *buses*. These are nothing more than sets of leads which are named for the sort of signals which they carry.

The function of the *data bus* for example, is almost self-explanatory.

In Chapter 1, we described the computer as a device which accepts numbers from the outside world as inputs, operates on these numbers, and provides numbers to the outside world as outputs. The data bus is the set of leads over which these numbers travel.

Of course, if the uP is to handle numbers in an orderly sequence, it must be capable of indicating from where the number is to come or where it is to go. The *address bus* is the pathway by which the uP selects a particular source or destination. The uP applies a number (address) to the address bus. Each block (ROM, input port, or output port) which can exhange numbers with the uP examines that address. The device which corresponds to that address then either places a number on the data bus or accepts the number that the uP has placed on the data bus, depending on the nature of the device.

The nature (read or write) and timing of such operations is controlled by signals which the uP places on the *control bus*.

Our Basic System

Our basic system will contain essentially what's shown in Fig. 4-1 plus a power supply. The control panel, uP and ROM each occupy individual circuit boards. The input and output ports are together on one board. Each board is built and checked out separately before the system is assembled.

In our system, the control bus contains nine lines, the address bus contains sixteen and the data bus contains eight.

The data lines are labeled D7 through DO and the address lines are labeled A15 through AO. In each case, the line which is labeled O is the least significant line and the line which is labeled with the higher number is the most significant line.

Since there are sixteen address lines, up to 65,536 individual devices or locations in memory can be addressed. For reasons which will become apparent much later, we'll divide the 65,536 addresses into three groups. For ROM, we'll reserve those addresses in which both A14 and A15 are 1. For input and output devices, we'll reserve those addresses in which either A14 or A15 (but not both) is 1. Addresses in which both A14 and A15 are 0 are reserved for a type of memory which we won't discuss yet — so-called "random-access memory".

Chapter 5
The Power Supply and Backplane

The Power Supply

The first item of business to consider in building our system is the power supply. We'll need a variety of voltages to power the system, and these will be provided by on-the-board regulators. Of course, we still must provide unregulated power to the regulators and that is the subject of the first part of this chapter.

The circuit which we'll use is shown in Fig. 5-1. It provides three unregulated voltages: +10.5 and ±17 volts.

The circuit is straightforward and requires little comment. Each voltage is obtained by bridge-rectifying the output of an appropriate transformer. The 1000-uF capacitors charge to the peak value of the rectified AC wave. Since the transformers provide 7.5, 12.6, and 12.6 volts rms, the unregulated outputs from the supply are 10.5, 17 and -17 volts, respectively.

The 1 Ohm, five Watt resistors limit the surge of current which occurs when the supply is turned on and the capacitors are uncharged.

The 0.01-uF, 1-KV disk ceramic capacitors protect the diode bridges from transients on the power line. Ac is brought in via a 3 wire plug and each leg is fused.

Layout is not at all critical. In the prototype, the layout is "early rat's-nest".

Verifying that the power supply is working properly is a matter of applying the smoke test. Plug it in and see if it works! If good-quality components are used and the supply is wired carefully, there should be no problem.

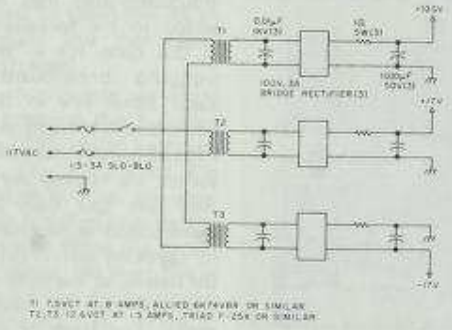


Fig. 5-1. Schematic diagram of power supply.

The Backplane

Our completed system will consist of a number of printed-circuit boards. In the prototype, each board is plugged into a socket on a master board, or *backplane*. It provides mechanical support and electrical connections for the boards. Photographs are shown in Figs. 5-2 and 5-3. It was made in two halves since I didn't have an etching tray which was large enough to hold the entire board.

None of the lines on the backplane (except ground) connect directly to TTL ICs. Because of this, it's possible that a "backplane" could be made simply by bolting the sockets to runners for mechanical support.

Hookup wire would then provide electrical interconnections. Indeed, we might dispense with the backplane and sockets altogether by stacking the individual boards (spacers between) and using hookup wire for electrical interconnections. Neither of these alternatives has been checked out, however.

Details of the backplane are shown in Fig. 5-4. No printed circuit layout is provided, however. If printed circuit boards are made by photographic techniques, it's a simple matter to lay out a network of tape on a mylar or acetate sheet and use that directly as the negative. The prototype was produced in exactly that way.

Verifying that the backplane works properly is a matter of checking for shorts between the various lines. The wise builder will check for shorts after each socket is soldered in place rather than waiting until all have been added, though. It's a very frustrating chore to remove three or four sockets trying to find the one that's causing a problem!

LINE	ASSIGNMENT
1	
2	GROUND
3	
4	A10
5	A11
6	A12
7	A13
8	A14
9	A15
10	D7
11	D6
12	D5
13	D4
14	D3
15	D2
16	D1
17	D0
18	READ/WRITE
19	SET OVERFLOW
20	Ø2
21	RESET
22	READY
23	Ø1
24	INTERRUPT REQUEST
25	NON-MASKABLE INTERRUPT
26	SYNC
27	BLANK
28	BLANK
29	A0
30	A1
31	A2
32	A3
33	A4
34	A5
35	A6
36	A7
37	A8
38	A9
39, 40	-17 VOLTS
41, 42	+17 VOLTS
43, 44	+5 VOLTS (REGULATED)
45, 46, 47	+10.5 VOLTS

Fig. 5-4. Line assignments for backplane.

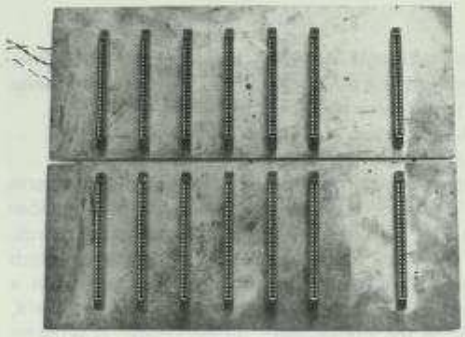


Fig. 5-2. Top of backplane.

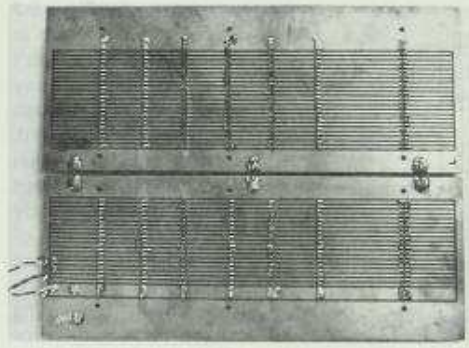


Fig. 5-3. Bottom of backplane.

The Debugging Breadboard

A handy device for troubleshooting may be added to the backplane at this time. It's a "debugging breadboard". Each signal line in the backplane is brought to a ProtoBoard® QT59S socket, via hookup wire. Any line can then be manipulated as necessary in checking out individual boards, or the state of any line can be monitored. The latter is accomplished by providing a 4049 inverter-buffer or two to drive individual

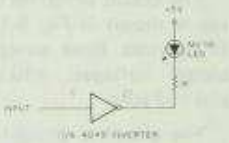


Fig. 5-5. LED monitor.

LEDs, as shown in Fig. 5-5. Regulated power can be thieved from the system, or a 5 volt zener diode can be used to adequately regulate the unregulated power.

Chapter 6
The Control Panel

We've finally reached the point where we can begin to build the interesting part of our system. In this chapter, we'll assemble and test the control panel.

Necessary Functions

The control panel serves three functions:

1. Provides control signals to the uP.
2. Provides the means to tie a small memory into the system.
3. Provides regulated +5 volt power for the basic system.

We'll examine each function in detail, starting with the control signals.

Until we've studied the uP in detail, the exact reasons why the control signals must have the properties which they do may be a little unclear. For the time being, however, let's concentrate on the what rather than the why.

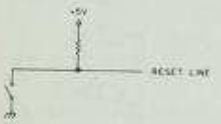


Fig. 6-1. Reset switch.

The first signal that the control panel must provide to the uP is the reset signal. The proper signal on the reset line will cause the uP to stop whatever task it may be doing and begin the execution of a series of instructions which start at a prespecified location. For normal operation, the control panel holds the reset line high. To reset the uP, the line must be taken low for an instant and then taken high again.

In principle, we could accomplish this by using an SPST switch as shown in Fig. 6-1. When the switch is open, the reset line is pulled high by the resistor. When the switch is closed, the reset line is pulled low by the closed contacts. In practice, we'd find that as the switch is thrown, a series of pulses result, because the contacts bounce for a millisecond or so. In this particular case that wouldn't matter, but generally we'll need switches which provide "bounceless" action. Since it's easily implemented and useful in checking out the remainder of the control panel, we'll do so even in this case.

The device which we'll use to accomplish this is

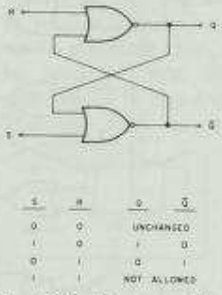


Fig. 6-2. Reset-set flip-flop and truth table.

called a reset-set (RS) flip-flop. A block diagram and truth table for our version is shown in Fig. 6-2. The characteristic which is useful to us is that if a series of pulses is applied to one or the other of the inputs, only the first pulse will produce an effect.

A schematic of the circuit which we'll use to provide the reset signal is shown in Fig. 6-3. If we examine it, we'll see that the logic levels are consistent and that the circuit is in a stable state. One input to the upper gate is held low by a pull-down resistor, while the other is held low by the output of the lower gate. Thus, the output of the upper gate is high. One input to the lower gate is held high by the application of +5 volts above the resistor, while

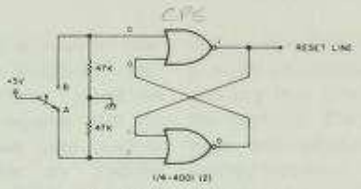


Fig. 6-3. Reset circuit.

the other is held high by the output of the upper gate. Thus, the output of the lower gate is low (as it must be to be consistent with what's said above).

Once the circuit is in this state, we can remove the +5 volts from point A without affecting the output of the circuit. In fact, we can do whatever we like at point A with no effect on the output, since the lower NOR gate will remain in the same state as long as either of its inputs is high. However, if we then apply +5 volts at point B (+5 volts removed from point A), the result is quite different. The output of the upper gate will go low, since one of its inputs has been taken high. Both inputs of the lower gate will then be low, hence its output will be high. The circuit is then in a second stable state with the output of the circuit low. Now we can do anything we like at point B with no effect on the output of the circuit. In short, the circuit provides the "bounceless" switching action which we need. The circuit is used as-is to control the reset line.

The second signal that the control panel must provide to the uP is the ready signal. When the ready input of the uP is taken high, the uP executes instructions one after another, at full speed. When the ready input is taken low, the uP stops and does nothing (at least in terms of executing instructions).

Unlike the reset input, the state of the ready input must be changed only in synchronization with the internal timing of the uP. For this purpose (and others) the uP makes available a square-wave, labeled Ø1. A basic ground rule for the 6502 uP is that the state of the ready input should be changed only when Ø1 is high. This suggests that we can use an RS flip-flop to control the ready line, by routing Ø1 to one or the other inputs of the flip-flop, depending on whether we want to take the line high or low. It further suggests that if we can use one Ø1 pulse to take the ready line high and the next Ø1 pulse to take it low, then we can work our way through a set of instructions one step at a time. In this way, we can see in detail how a given operation is accomplished.

Our black-box for this portion of the control panel, then, accepts contact closures from a push-button (single-step) and a toggle switch (run-stop), and a pulse train (Ø1, square wave) as inputs. The output is a logic level which depends on the states of the switches and which changes only in synchronization with the pulse train. A circuit which will accomplish this is shown in Fig. 6-4.

In the lower portion, we see an RS flip-flop which is driven by Ø1, via a pair of NOR gates connected as inverters. (Since the toggle switch which routes Ø1 may be located off the board, we use the inverters to buffer the Ø1 pulse train before it leaves the board). The output of the RS flip-flop is applied to one input of a NOR gate which, in turn, drives an inverter. The output of that inverter controls the ready line.

Regardless of what happens in the part of the circuit which we haven't discussed, if the toggle switch is thrown to run, the ready line will be taken high. If it's thrown to stop, the line may be taken low, but whether it is or whether it stays that way depends upon the remainder of the circuit.

At the upper left of Fig. 6-4, we see an RS flip-flop which is driven by +5 volts via a

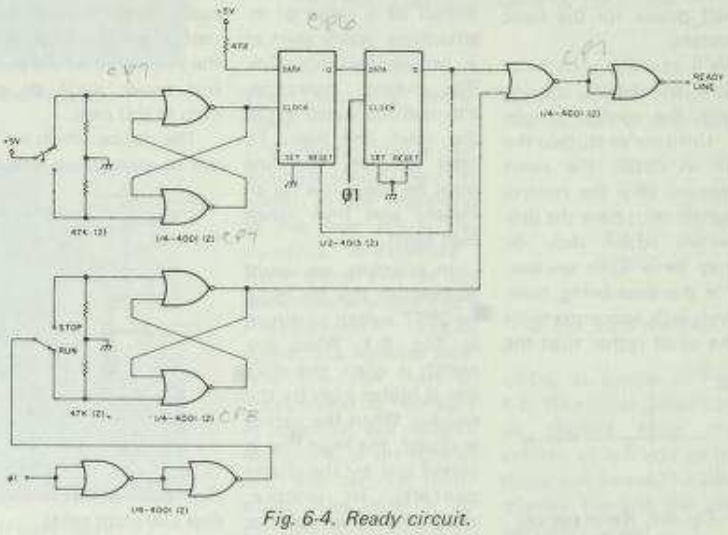


Fig. 6-4. Ready circuit.

momentary-contact SPDT push-button. If we push and then release the push-button, we generate a positive-going pulse at the output of the RS flip-flop. This positive-going pulse is applied to the clock input of a 4013 positive-edge-triggered type D flip-flop. Since +5 volts is applied to the D input of the 4013, a high level appears at its Q output in response to the positive-going edge of the pulse which was generated by the RS flip-flop. In turn, this high level is applied to the D input of the second 4013. The second 4013 is triggered by the positive-going edge of the next Ø1 pulse, so that the high level is passed from the D input to the Q output of that 4013. This does two things. First, it takes the ready line high. Second, it resets the first 4013. This takes the Q output of the first 4013 low. In turn, that low level is applied to the D input of the second 4013. The positive-going edge of the next Ø1 pulse triggers the second 4013, passing on the low level to the Q output of that 4013. Unless the toggle switch has been thrown to run, this takes the ready line low. A moment's reflection will indicate that the output of the second 4013 will remain low until the single-step push-button is activated again.

The net result is that, after the single-step push-button has been depressed, the first Ø1 pulse takes the ready line high and the next takes it low again.

Link to Small Memory

As we've discussed, in the next chapter we'll construct a small ROM for our basic system. To tie this memory into the system, we must connect it to the eight data lines and eight ($256=2^8$) lower order address lines, and provide a signal to control the transfer of numbers from it. Tying it to the address bus and the data bus is no problem. We simply provide tie points on the control panel which are connected to the buses. Providing a control signal is not much more difficult. What's required is a signal which indicates that the uP expects to read the contents of a location within the ROM.

One output of the 6502 is the *read/write* (R/W) line. If the 6502 expects to read data from the data bus, it takes this line high. Clearly, the control signal to the ROM must take this into account. Further, in Chapter 4, we adopted the convention that high levels on address lines A14 and A15 indicate that a ROM is involved. Clearly, the control signal to the ROM must also take account of the states of lines A14 and A15.

The ROM transfers data to the data bus via solid-state switches. A high level to the control input of the switches turns them on. One way to provide a control signal to the ROM, then, is to AND the R/W line with lines A14 and A15. This will turn on the solid-state switches when the uP expects to read from a location within the ROM. If we don't intend ever to expand the ROM beyond 256 words, this is an acceptable way to generate the control signal. However, we'll want to consider expansion in a later chapter. To accommodate that possibility, we'll include the remaining address lines (A8 through A13) in the circuit, but we'll hedge a bit regarding how they're included. The circuit is shown in Fig. 6-5.

As we'd expect, the R/W line and address lines A14 and A15 are ANDed (actually they are NANDed and the result is inverted). Address lines A10 through A13 or their complements in any combination are also applied to the inputs of the 74C30 NAND gate. If a nine-input NAND gate were available, we could also apply A8 and A9 directly. Since one is not, we combine and apply the latter two signals or their complements (in any combination) by means of the remaining portion of the circuit.

Until we've gone beyond the basic system, the jumpers should be placed as shown in Fig. 6-5. In that way, the ROM is selected when address lines A8 through A15 (and the R/W line)

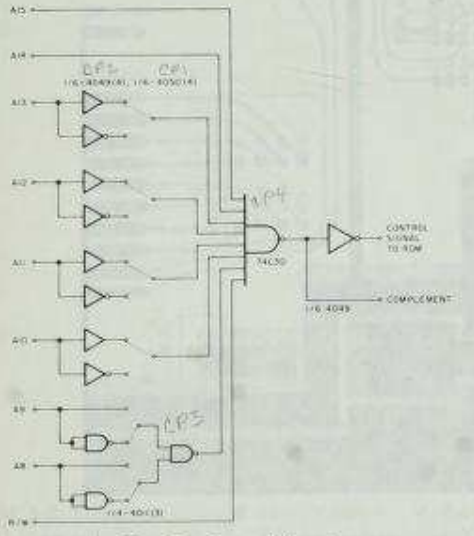


Fig. 6-5. Control circuit.

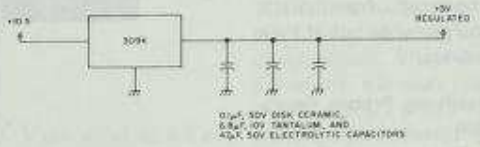


Fig. 6-6. Power supply.

are all high.

The complement of the control signal is also provided. As we'll see later, this will be useful if the system is expanded.

Power Supply

The final function which the control panel serves is to provide +5 volt regulated power for the basic system. A type 309K regulator is used in the circuit which is shown in Fig. 6-6. Disk ceramic, tantalum, and electrolytic bypass capacitors are used. In the prototype, the regulator was unstable to a degree unless all were provided.

Construction

The foil side of the PC board and the component layout are shown in Figs. 6-7 and 6-8, respectively.

Those who choose one or the other alternatives to PC board construction shouldn't have any major problems. During the design of the prototype, all the functions of the control panel were breadboarded. The only problems involved were due to poor connections, solder blobs, and the like.

Regardless of the method of construction, sockets should be used for all CMOS ICs. Power should be off when the ICs are removed or inserted, and they should be protected from static charges when out of their sockets.

Verifying Proper Operation

The on-board regulator should be tested first. We

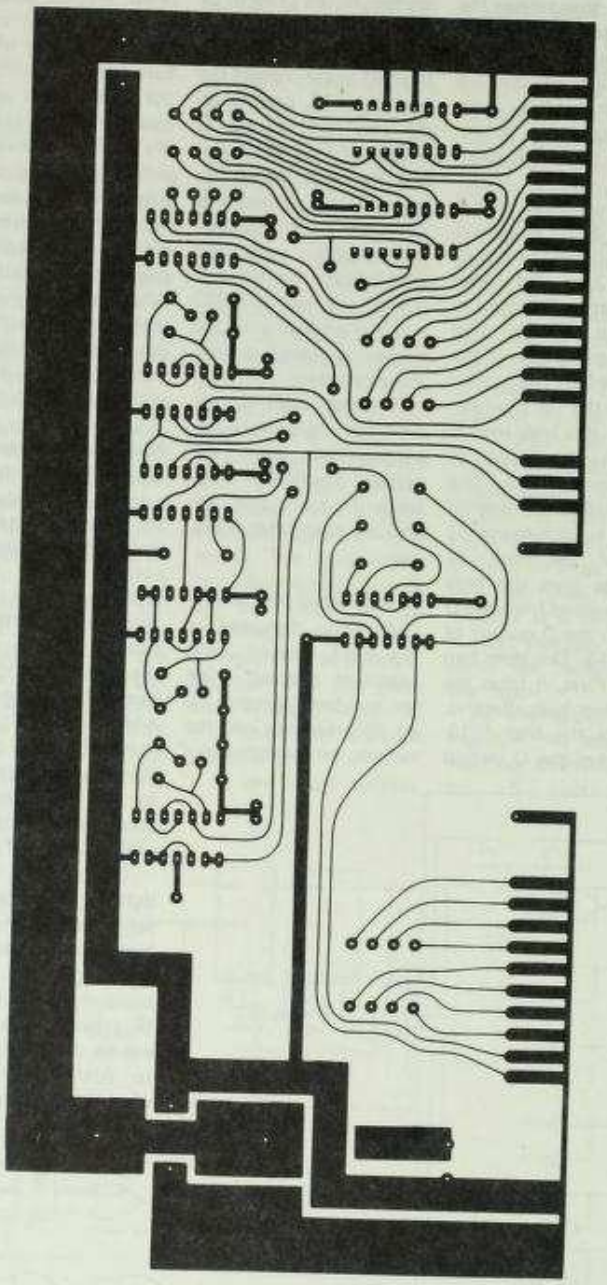


Fig. 6-7. Foil side of the control panel.

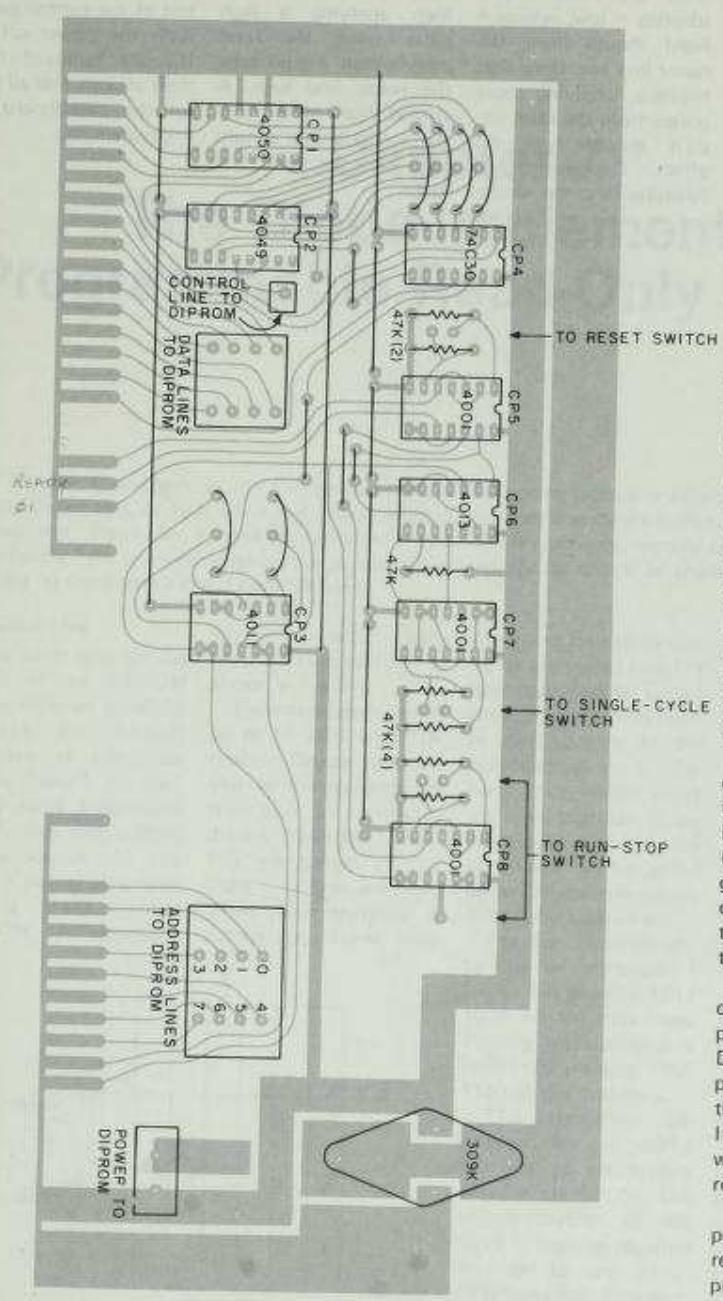


Fig. 6-8. Component layout, control panel. Tie .1 uF, 50 V disc ceramic; 6.8 uF, 10 V tantalum; and 47 uF, 25 V electrolytic capacitors between regulated +5 bus and ground at regulator. Use heatsink on regulator.

can best do this by plugging the control panel into the backplane, applying power, and seeing if the regulator produces +5 volts at its output. Of course, for this test all other ICs are removed.

Once the power supply is working properly, we can test the circuit which provides the control signal to the ROM. For this test, ICs CP1, CP2, and CP4 should be in their sockets. Making use of the debugging breadboard, we program address lines A8 through A15 and the R/W line high. This should cause the control output to go high and its complement to go low (logic probe or LED monitor). If we reprogram any of the lines low, the control signal should go low and its complement should go high.

To complete the checkout of the link to the ROM, we remove the ICs from their sockets (power off), "unprogram" the lines, and then check for continuity between each tie point and the corresponding line.

Next, we test the reset circuit. With IC CP5 in place, we apply power. Depressing the reset pushbutton should cause the reset line to go low. It should go high again when the pushbutton is released.

If all is well at this point, we can test the ready circuit. With the power off, the reset line and Ø1 line are jumpered together at the debugging breadboard. This allows us to use the reset circuit

to simulate the Ø1 pulse train. ICs CP6, CP7, and CP8 are inserted in their sockets, and power is applied.

When the power is turned on, the ready line may be either high or low. A high pulse or two from the reset circuit (depress = low, release = high) should bring the ready line low. Once that happens, applying more pulses from the reset circuit should have no effect. Depressing and releasing first the single-step push-button and then applying a high pulse using the reset push-button should take the ready line high. A second high pulse from the reset circuit should then take the ready line low.

That completes the test of the control panel. With the power off, all ICs are removed from their sockets and all lines on the breadboard are unprogrammed.

Chapter 7
The Diode-Implemented Programmable Read-Only Memory

In Chapter 1, we characterized the typical small system as being tedious or expensive to communicate with. We concluded that, at least while our system is small, we'll use a somewhat unusual device to overcome this shortcoming. The device, a diode-implemented programmable read-only memory (DIPROM), is the subject of this chapter. It's a memory which we program mechanically by means of small PC cards, each of which contains no more than a single eight-bit word.

The Basic Idea

The basic idea for the design of our DIPROM comes from an excellent article by Robert Cushman in *Electronic Design News*.[1] He describes some preliminary experiments with a 6502 microprocessor and presents a partial schematic for a sixteen-word DIPROM. A schematic which presents Cushman's suggestion is shown in Fig. 7-1.

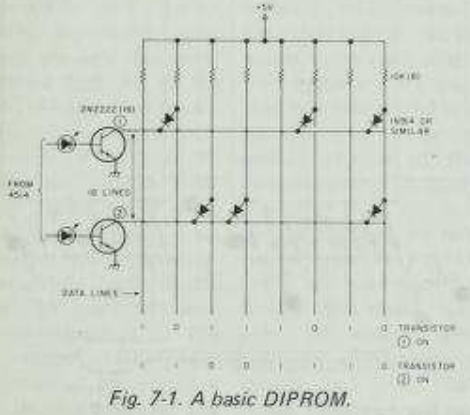


Fig. 7-1. A basic DIPROM.

Each data line is pulled up to +5 volts via a 10k resistor. Certain lines are tied to the collectors of transistors by means of diodes. When the transistors are turned off, all eight data lines are high. When either transistor is turned on, those lines which are tied to its collector are pulled low. It's important that we understand that the lines can't be tied directly to the collectors of the transistors. That would short the lines together. If we use diodes as shown, the data lines can be pulled down by the transistors, but not by each other.

For the circuit shown, turning on transistor 1 (only) will produce 1011 1010 on the data lines. Turning on transistor 2 (only) will produce 1100 1110 on the data lines.

The transistors are driven by a 4514 four-line to sixteen-line decoder via LEDSs. The unique output of the 4514 is high, as required in order to turn on an NPN transistor. By inserting LEDs in the lines to the transistors, it's possible to see which transistor is turned on at any time.

If we apply a 4-bit word to the input of the 4514, we produce an 8-bit output on the data lines. The particular output which is produced depends on the particular transistor which is turned on and the pattern of diodes which is associated with than transistor. If the diodes are placed on small PC plug-in cards, the patterns can be changed easily.

Improvements On The Basic Idea

The first DIPROM which was used with the prototype of our system was essentially as shown in Fig. 7-1. While it was very useful for getting the system up and running, it was too small to do much more. Then too, it was too slow.

That is, an excessive amount of time elapsed between the application of an input and the appearance of the corresponding output. Accordingly, I set about to enlarge and speed up the DIPROM.

The speed of the DIPROM is related to a number of things — among them being the distributed resistance and capacitance in the circuit. For example, decreasing the resistance of the pull-up resistors to 3k from 10k increases the speed of the memory by a significant amount. In fact, that was the first change which was made to the prototype.

The speed of the DIPROM is also related to the times that the various devices take to change state. For example, as much as 400 nanoseconds may elapse from the time that the input of the 4514 is changed until the output changes. Not much can be done about that, since it involves the internal workings of the 4514. Similarly, once we've selected a transistor, we can't do much about its internal switching characteristics. However, we can make sure that we use such devices in the best way. For example, some tests on the prototype indicated that the transistors apparently were being turned on very hard and, because of that, were slow in turning off. Each LED was replaced with a resistive voltage-divider network, and this improved the switching action. (The LEDs probably could be used in conjunction with the networks, but this was not tried).

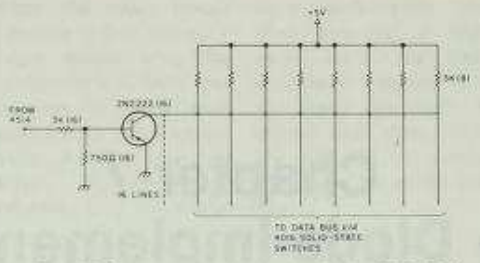


Fig. 7-2. Sixteen word subunit of the DIPROM.

These changes produced the basic sixteen-word subunit for our DIPROM. A schematic is shown in Fig. 7-2.

A Larger DIPROM

At this point we still have just a sixteen-word memory. We could build, say, eight such memories, producing a capacity of 128 words. However, we'd pay dearly for it. A single 4514 decoder alone costs $5 at the time that this is written.

There is a less expensive way, though, as shown in Fig. 7-3. Here we've taken four of the sixteen-word subunits and combined them into a sixty-four-word array. Each of the four subunits is driven by the same set of sixteen transistors. Depending on which of these transistors is turned on, the corresponding words will be selected in each of the four sixteen-word subunits. Our task then becomes to select the particular one of the four words which is to be passed on to the output.

We do this by means of the circuit which is shown below the sixteen-word subunits in the figure. In effect, we've added a seventeenth word to each sixteen-word subunit. Each bit of each added word is set to 0. If we turn on three of the four transistors in the lower set, the corresponding words will be driven to 0000 0000, regardless of what happens within the subunits. That is, we've disabled the outputs of the three subunits in terms of their effect on the NOR gates. The output of each NOR gate will then depend only on the word which

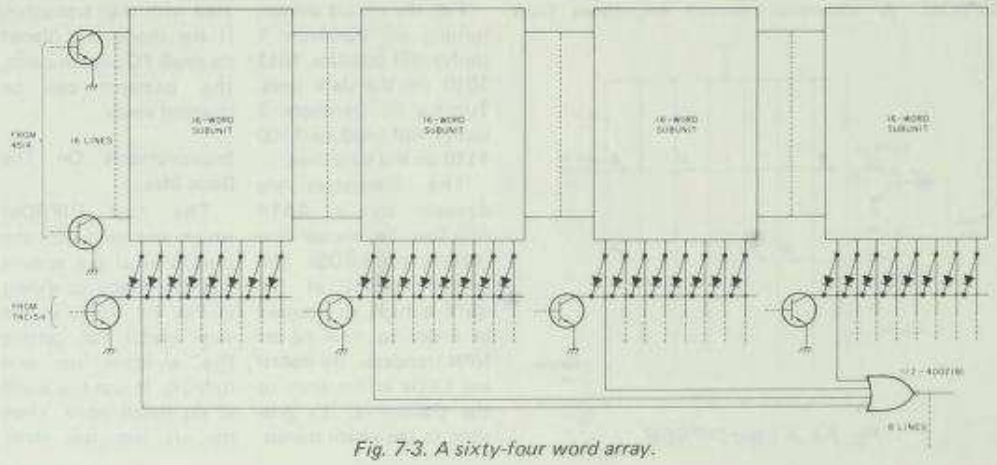


Fig. 7-3. A sixty-four word array.

appears at the output of the fourth subunit.

Because we've used NOR gates, the output of the sixty-four-word subunit will be the complement of the selected word. But that's not a problem, as we'll see in a moment.

The decoder which we use to drive the second set of transistors must produce a unique output low, since only one transistor is turned off at any time. The 74C154 decoder is such a device.

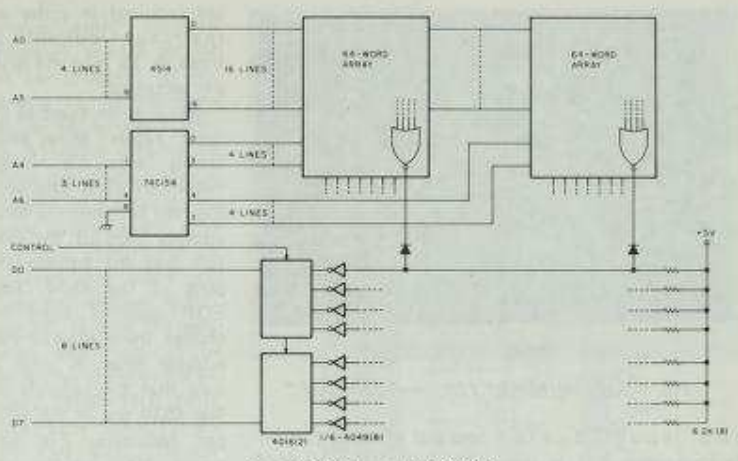


Fig. 7-4. 128 word DIPROM.

Because of the size of the printed circuit boards which are involved, the sixty-four-word unit is a convenient basic unit. Of course, the 74C154 has sixteen outputs so that we can combine up to four of the sixty-four-word units, producing up to a 256-word memory. The way in which two sixty-four-word units are combined is shown in Fig. 7-4. Expansion to four units is straightforward.

Address lines A0 through A6 are applied to the two decoders as shown. If four sixty-four-word units were use,we'd also apply address line A7 to the 74C154 and drive the additional two units with outputs 8 through 15 from the 74C154. However, since we're using only two units, we ground the highest-order-address input of the 74C154 and don't connect outputs 8-15. The remainder of the upper portion of the circuit is as described before. The output of one of the sixty-four-word units will be the complement of the word which should appear at the output of the circuit. The output of the other sixty-four-word unit (in which the outputs of all the sixteen word subunits are disabled) will be 1111 1111.

If we now turn Fig. 7-4 on its side so that the decoders are at the bottom, we see a somewhat familiar sight at the right of the figure. We have eight data lines each of which is pulled to +5 volts via pull-up resistors. Each data line can also be pulled down via a diode. This happens if the logic level which is appled to a diode is taken low by the output of one of the sixty-four-word units. At any given time, one unit will be disabled. Its output will be 1111 1111 and it will have no effect on the data lines. However, the enabled unit will have an effect. If its output is, say, 1100 1010, then that will appear on the data lines. Out of the collection of 128 words, the desired one, or at least its complement, has been located and placed on the data lines.

Because the output of

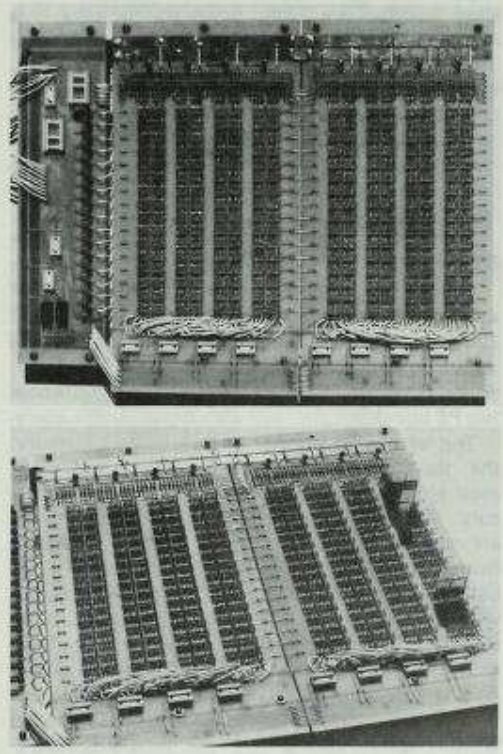


Fig. 7-5(a). Top of 128-word DIPROM (two views).

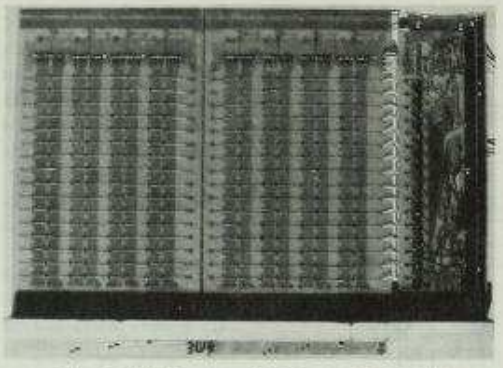


Fig. 7-5(b). Bottom of 128-word DIPROM.

the enabled sixty-four-word unit is the complement of the desired word, we apply the data lines to inverters which, in turn, drive solid-state switches. The other sides of the solid-state switches are connected to the data bus.

The switches are turned on by application of a high level to their control inputs. A low level turns them off, disconnecting the memory from the data bus.

Construction

Photographs of the top and bottom of a 128-word DIPROM are shown in Fig. 7-5. The device is constructed on five PC boards.

The leftmost board in the top view contains most of the switching circuit. The two decoders are near the top of the board. The sixteen transistors which select individual words within the sixteen-word subunits are in a column at the right. At the very bottom are the 6.2k pull-up resistors. Directly above them are the two 4049 inverters and just above the inverters are the two 4016 solid-state switches. Power leads connect to the board at the upper left. The address lines connect just below the power leads. Further down are the data lines and the control line for the solid-state switches. The collector leads for the sixteen transistors leave the board at the left and the data lines to the inverters come in from the bottom left.

The careful observer will note that there's an IC on the upper-left of the narrower PC board about which nothing has been said. It serves no fundamental purpose with regard to the operation of the memory. However, including it makes the layout of the circuit which involves the 4514 decoder somewhat less complicated.

If we examine the pin connections of the 4514, we'll see that the outputs aren't in a convenient sequence for our purpose. Outputs 0 through 7 are arranged reasonably, but several jumpers are required in order to rearrange outputs 8 through 15 into the proper sequence.

More as an exercise in logic rather than anything else, a quad two-input EOR gate was inserted between the four address lines of the system and the address inputs of the 4514. The EOR gate is used to change the coding of the address lines in such a way that the outputs of the 4514 are in the proper sequence. For example, 1000 on the address lines produces 1101 at the inputs of the 4514.

Each of the two large boards forms the major portion of a sixty-four-word unit. The wide strips at the top carry power. Just below these are the transistors which enable or disable entire sixteen-word subunits. The pull-up resistors for the individual subunits are just below the transistors.

The diodes which form the seventeenth words in each of the sixteen-word subunits are located at the bottom of the large boards.

On the small boards at the bottom are the dual four-input NOR gates, and just below the gates are the diodes which tie the outputs of the NOR gates to the on-board data lines.

The major portion of each larger board is occupied by the sockets into which the individual words are plugged. Each socket consists of ten Digiclips®. From left to right, the first four connect to four data lines. The fifth connects to the collector of the appropriate transistor (under the board). The next four connect to the remaining four data lines, and the tenth connects to the collector of the (same) appropriate transistor.

Using this configuration, our basic plug-in unit is a four-bit word. A number of the four-bit plug-ins are shown in place at the far right of the second large board and a few are shown in more detail in Fig. 7-6.

The spacing within the

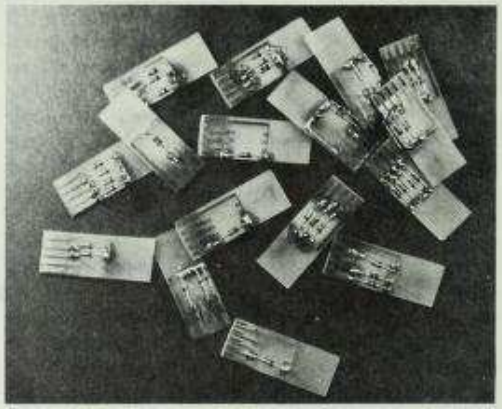


Fig. 7-6. Four bit plug-ins.

PC layout for the plug-ins is such that eight-bit plug-ins (two four-bit plug-ins side by side) will fit directly into the sockets. Since instructions for the 6502 are made up of eight-bit words, using eight-bit plug-ins means that they can be labeled directly with the abbreviations of the instruction which they implement.

Constructing a 128-word memory requies soldering 1280 individual Digiclips® into place. The best way to do this is to use a jig such as the one shown in Fig. 7-7. It's a pair of unseparated four-bit plug-ins. Ten small notches are cut at the bottom so that ten Digiclips® can be loaded on to the jig and be held in place reasonably well. The whole works is then put in place on the board and each Digiclip® is soldered to the board.

No holes should be drilled for the Digiclips®. They should be soldered directly to the top of the PC foil.

Digiclips® may not be conveniently available (however, see the section on construction in Appendix 1). In that case, Molex Soldercons® may be a useful alternative. These are single-pin connectors on a retainer. As many as are needed (five at a time, in our case) are soldered into place and then the retainer is broken off. If they are used, holes will have to be drilled for them in the PC board, AND THE FOIL PATTERN SHOULD BE ON THE BOTTOM OF THE BOARD,RATHER THAN THE TOP. They would also have to be used as "fingers" on the plug-ins.

The foil side of the PC boards and the component lay-outs for each are shown in Figs. 7-8 through 7-15 (*Editor's note: Due to their large size, Figs. 7-8 and 7-10 appear as foldouts in the center of the book.*).

There doesn't appear to be a good alternative to a PC board for the portion of the memory into which the plug-ins are inserted. However,

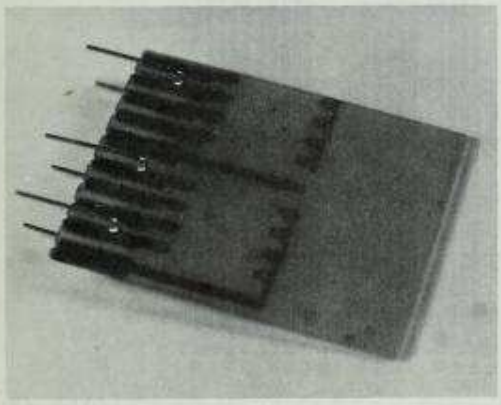


Fig. 7-7. Holder for DIGICLIPS®

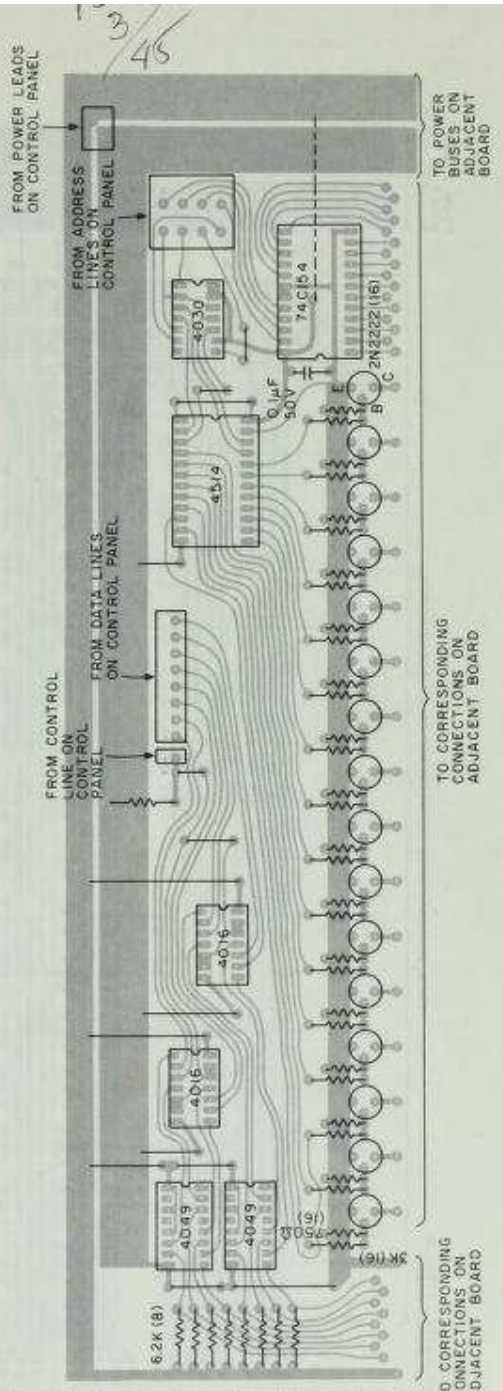


Fig. 7-9. Component side of input/output board, DIPROM. Dotted line indicates jumper under board.

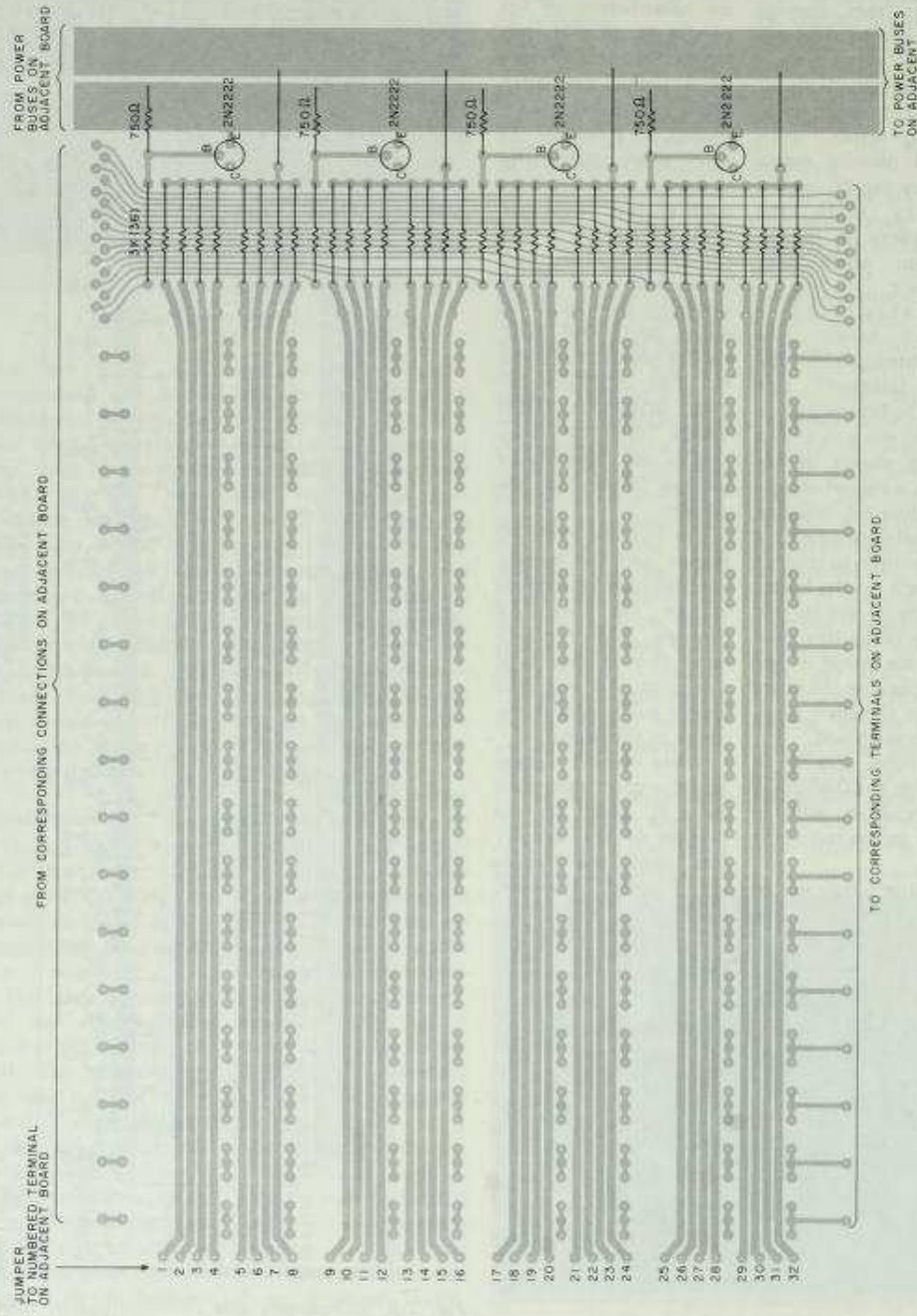


Fig. 7-11(a). Components on foil side of matrix board, DIPROM.

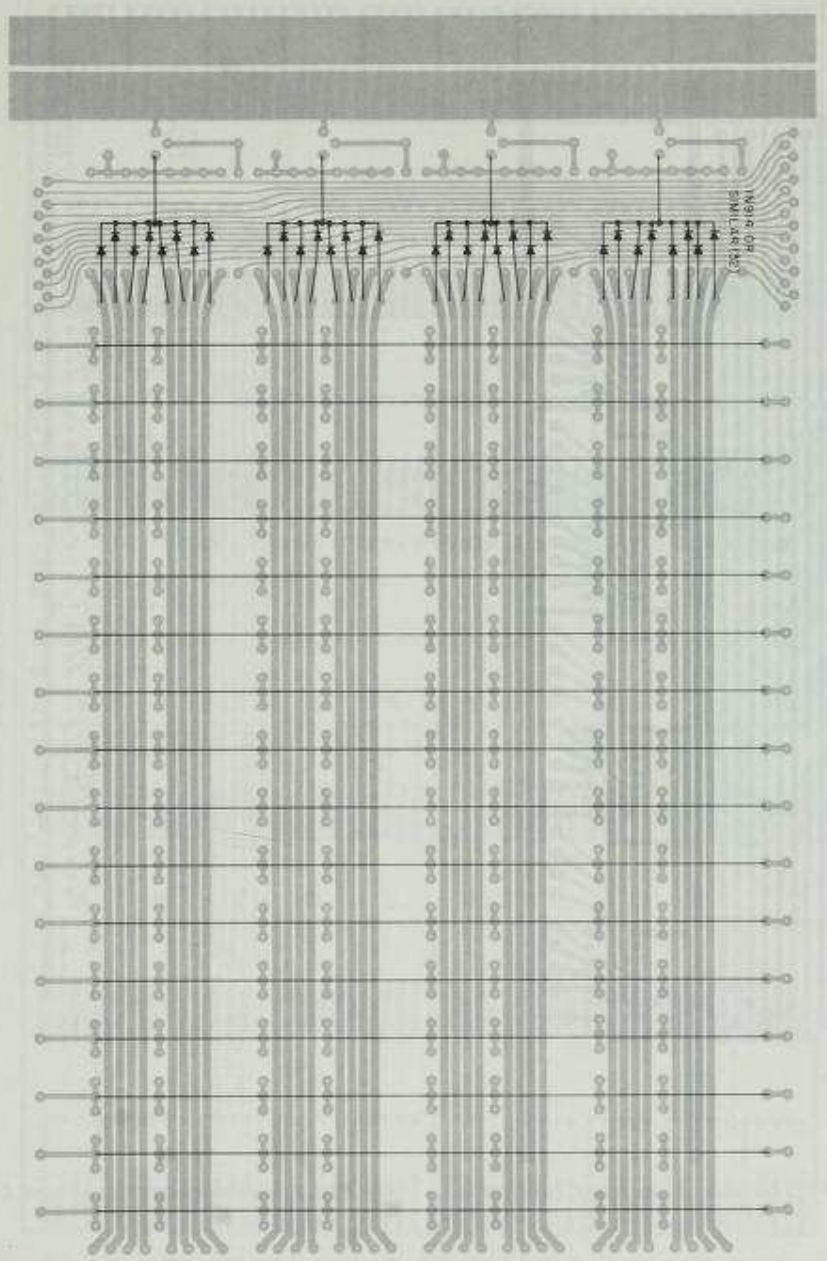


Fig. 7-11(b). Components on non-foil side of matrix board, DIPROM. Diodes at upper end of board are wired together in groups of eight and connected to the collector of the appropriate transistor. See Fig. 7-5(b).

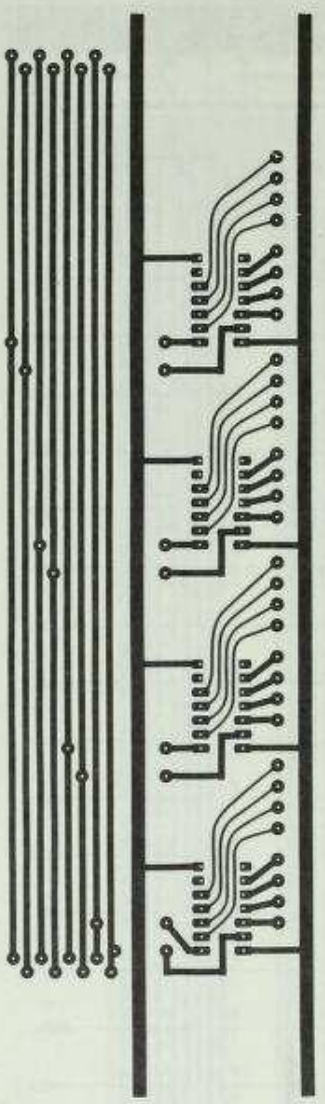


Fig. 7-12. Foil side of data board, DIPROM.

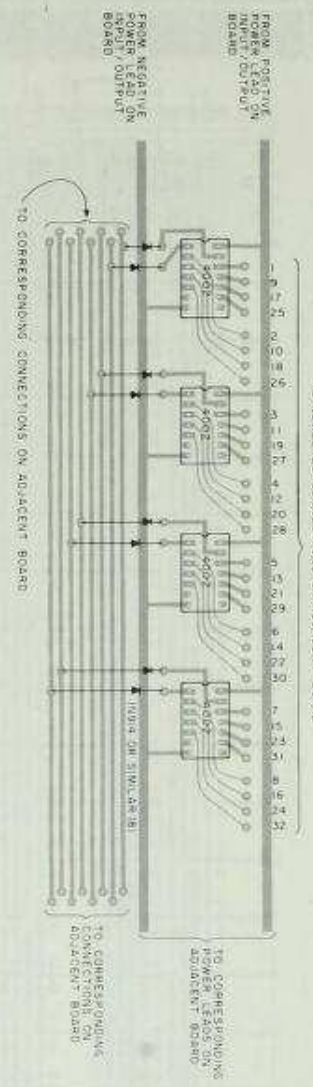


Fig. 7-13. Component side of data board, DIPROM.

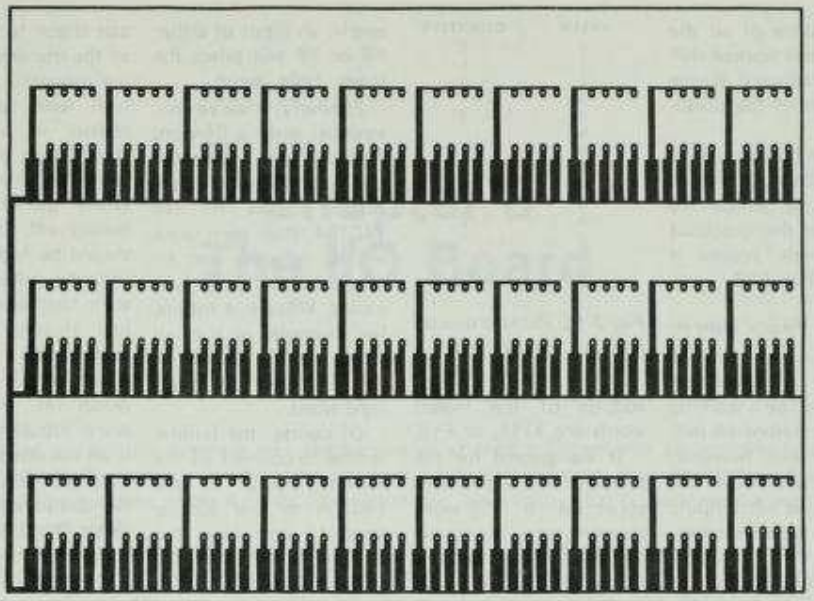


Fig. 7-14. Foil side of plug-in units for DIPROM. Fabricate from 1/32" PC stock.

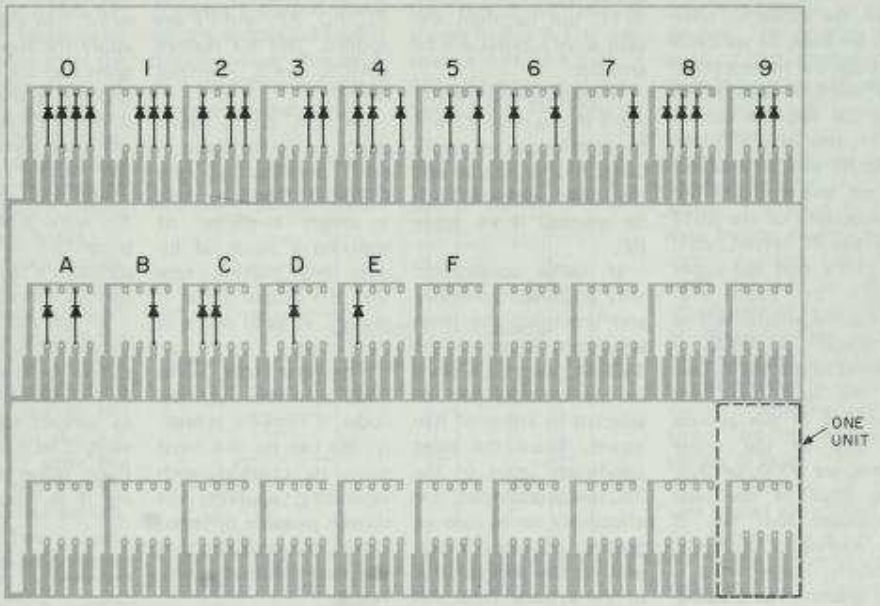


Fig. 7-15. Component layout of plug-in units for DIPROM. Diodes are soldered to foil side of board. All diodes are 1N914 or similar. Note that no "F" boards need be made.

the functions of all the other boards worked well on a breadboard during the design of the prototype.

For the builder who is looking ahead, a list of the plug-ins which are needed in the checkout of the basic system is shown in Fig. 7-16.

VALUE	QUANTITY
0	5
1	1
2	2
3	–
4	2
5	–
6	–
7	–
8	2
9	1
A	2
B	–
C	1
D	2
E	1

Fig. 7-16. Plug-ins needed to check basic system.

Verifying Proper Operation

At this point, we have what should be a working DIPROM. Before we perform any tests, however, it's important that we understand just what inputs will select what locations, and why.

Regardless of the size of the memory (assuming that we use at least one sixty-four word unit), all the address inputs of the 4514 decoder will be used. As should be obvious by now, as we cycle through the sixteen possible combinations which can be applied to the 4514, one of it's sixteen outputs will be enabled. If we examine the pin connections of the 4514 and the PC layout, we'll conclude that the upper word in each sixteen-word subunit will be activated if 0000 is applied to the 4514. That is, the least significant four bits of the address of any of the upper words are 0000, or 0_{16}. The value of the least significant four bits of the address of each successively lower word is one greater. The least significant four bits of the address of the lowest words are 1111, or F_{16}.

If we assume for the moment that we've constructed a 256-word memory, then by studying the pin connections of the 74C154, we'll conclude that the far left sixteen-word subunit will be enabled if we apply 0000 to the 74C154. Similarly, if we apply 1111, the far right sixteen word subunit will be enabled.

In short, the upper left word in the memory will be selected if we apply 00_{16} to the inputs, and the lower right word will be selected if we apply FF.

If we've constructed only a 128-word memory and grounded the most significant input of the 74C154, then each word in the memory can be selected by either of two inputs. Since the most significant input of the 74C154 is grounded, it is effectively set to zero regardless of what is present on the address lines of the system. What this means is that, for example, an input of either FF or 7F will select the lower right word.

Similarly, if we've constructed only a 64-word memory and grounded the two higher-order address inputs of the 74C154, then each word in the memory can be selected by any of four inputs. What this means, for example, is that an input of FF, BF, 7F or 3F will select the lower right word.

Of course, the builder is free to connect all the address inputs of the 74C154 to the address lines. In any case, the jumpers which pass the outputs of the 74C154 to the sixty-four-word unit(s) should be arranged so that locations are available which are selected when FA, FB, FC, FD, FE, and FF are applied. This for reasons which we'll consider later.

At this point, we're ready to verify the proper operation of the DIPROM. Basically, this is simply a matter of applying a series of inputs and making sure that the proper signal is present at each point in the circuit.

The first task is to make sure that each decoder is working properly. We can do this most easily by checking each separately, applying the sixteen possible different inputs and verifying that the proper output is activated.

At the same time, we can check to see if each of the transistors is working properly. Applying a high level to the base resistor of a transistor should turn it on. A low level should turn it off. When the transistor is turned off, its collector should be high, provided that the collector is tied to a high level, and not just floating. The collector of a transistor which drives individual words in the sixteen-word subunits will float if all the words which the transistor drives are FF. No comparable situation exists if the transistor is turned on. In that case, the collector is low.

If the decoders and transistors are working properly, our next step is to insert the remaining ICs (power off), plug a word into the DIPROM, apply the appropriate address and see if the word appears at the output. If it doesn't then we work our way backward from the output to see at which point in the circuit the word is present. This procedure will serve to pinpoint a faulty component or connection.

In all this we should remember to turn off the power before removing or inserting any ICs, and to connect each input of every CMOS IC to something rather than allowing it to float.

Reference

[1] Cushman, R. H., *EDN*, 20, 117 (1975).

Chapter 8
The I/O Board

At this point we can put numbers into the system, in at least a limited way, by plugging tabs into the DIPROM. We have, in a sense, a mechanically operated input port. However, as we use even our basic system we'll need to enter numbers in the form of logic levels into the system. That is, we'll need an electronically operated input port.

Further, we'll need some way to get numbers out of the system. That is, we'll need an output port.

Finally, for reasons which we'll see later, we'll have use for a timer circuit.

These three functions are grouped together on one board, the I/O board, and that's the subject of this chapter.

Necessary Functions

For the input port, we need a circuit which will accept and (optionally) store a number from the outside world. The uP must then be able to read the number which is applied to, or stored in, the port. It's convenient to divide this total function into two black boxes.

The first black box accepts an eight-bit number from the outside world. Optionally, a signal in the form of a high or low level (operator selectable) may be applied to the black box from the outside world to store the number in the box. If that signal is not applied, the black box simply tracks the number at its input.

The output of the first black box is applied to the data bus in response to a control signal from a second black box. Since the second black box also provides a control signal for the output port, we'll postpone our discussion of it until after we've said more about that port.

A circuit which will provide the function of the first black box is shown in Fig. 8-1. It consists in part of a pair of 4042 four-bit level-sensitive latches. The eight-bit number from the external world is applied to the data inputs of the latches. The polarity inputs of the latches are tied to ground. Thus, if we apply a low level to the clock inputs, the latches will follow any changes in the number which is applied to the data inputs. If the clock input is taken high the latch will hold the value which existed at the time of the transition. Note that the data and clock inputs are tied to ground via 47k resistors rather than being allowed simply to float.

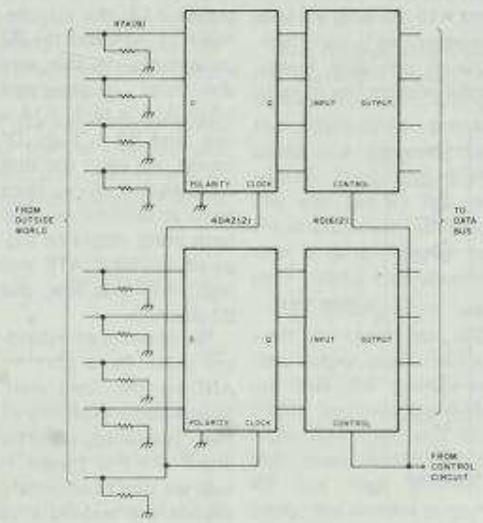


Fig. 8-1. Input port.

The outputs of the two latches are applied to the data bus via a pair of 4016 four-bit solid-state switches. The switches are turned on and the latches are tied to the data bus if a high level is applied to the control in-

puts of the 4016s. A low level turns the switches off, disconnecting the outputs of the latches from the data bus. For our basic system, we'll need two such input ports.

At this point, then, we have a way to get numbers into the system. Let's consider how to get numbers back out of the system.

For the output port, we need a circuit into which the uP can write a number and which can provide that number to the outside world. It's convenient to divide this function into two black boxes.

The first black box accepts an eight-bit number from the data bus. In response to a control signal from a second black box, the first stores the number. The eight-bit output from the first black box is available to the outside world. A circuit which will provide the function of the first black box is shown in Fig. 8-2.

It consists in part of a pair of 4042 four-bit level-sensitive latches. The data inputs of the latches are tied to the data lines. The polarity inputs of the latches are tied to ground. A negative-going control pulse is appled to the clock inputs of the latches when the number which is on the data bus is to be stored. The outputs of the latches are made available to the outside world via a pair of 4050 buffers. For our basic system, we'll need two such output ports.

At this point, we have four ports through which our system can exchange numbers with the outside world. We now have to provide a circuit which will allow the uP to operate the ports as it requires.

Inputs to the black box which will provide this function include (among others) the R/W line, the Ø2 pulse train, and address lines A14 and A15. As we'll see in a moment, we'll also dedicate an individual, lower-order address line to each

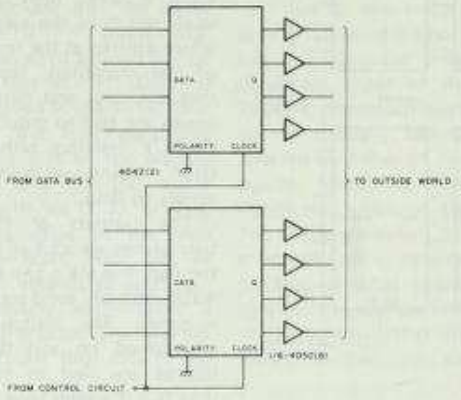


Fig. 8-2. Output port.

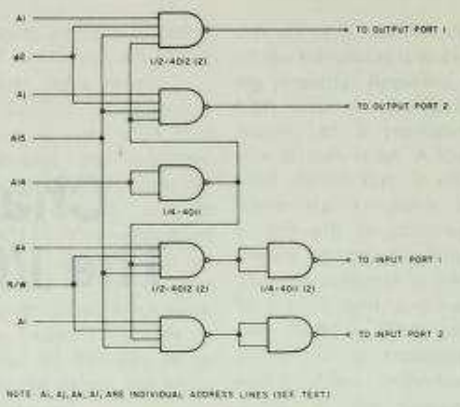


Fig. 8-3. Control circuit for I/O board.

port.

Outputs from the black box include a pair of negative-going pulses (one to trigger each output port) and a pair of positive-going signals (one to turn on the switches in each input port).

A circuit which will provide the function is shown in Fig. 8-3.

Each pulse is generated in part by ANDing together A15, the complement of A14 and the Ø2 pulse train. In this way the pulse is generated when A15 is high, A14 is low, and Ø2 is high. Of course, we can't use that combination alone, since to do so would mean that both ports would be triggered anytime A15 was high, A14 was low, and Ø2 was high.

We generate an individual pulse for a port by ANDing the lines mentioned above with one of the remaining address lines. All this means is that we don't completely decode the address lines in order to select an individual output (or input) port. For example, if we choose A9 as the line which we'll dedicate to one output port, then that port will be selected whenever the combination

10XX XX1X XXXX XXXX

appears on the address bus. The Xs indicate that the digits at those locations have no effect of the selection. Thus, any of the addresses which result from any combination of ones and zeros in place of the Xs will select the same output port. If we replace all the Xs with zeros, for example, the address is 8200. This is wasteful, at least in one sense, because we can't address any other input or output port (A15=1, A14=0) with an address in which A9 is 1. However, we don't have so many ports that it becomes a problem, and we simplify the control circuit by using this method.

The remainder of the circuit is similar. Individ-

ual, dedicated address lines are provided for each port.

One final point remains. We note that the R/W line is used in the generation of the control signals for the input ports, as we'd expect. However, it's not used in the generation of the pulses for the output ports. This is of no practical consequence. While we certainly don't want the uP to try to write into an input port – to do so would likely damage some logic elements – no comparable problem exists if the uP "reads" an output port. In practical terms, this means that only four inputs make up each pulse to an output port. Thus, we can use a single IC to generate both pulses. If we include the R/W line, two ICs are required, to accommodate the two sets of five inputs each.

The final portion of the I/O board is the timer. It's a device which provides an output that changes state in response to a pulse at its input and remains in that state for a predetermined time. At the end of the predetermined time, the output reverts to its previous state and remains there until another pulse is applied to the input.

To provide this function, we'll use a 4047 astable/monostable multivibrator in the circuit which is shown in Fig. 8-4.

The timer is set up for negative-edge triggering and is programmed to produce a quarter-second-long signal at the output. The trigger input is tied to +5 volts via a 1M resistor so that the input doesn't float.

Construction

The foil side of the printed circuit board is shown in Fig. 8-5. All the circuits discussed in this chapter are included. The component layout is shown in Fig. 8-6.

Those who chose one or the other alternatives to PC-board construction shouldn't have any major problems. During the design of the prototype, all the functions of the I/O board were breadboarded in "early rats-nest." The only problems encountered involved poor connections, solder blobs, and the like.

Regardless of the method of construction, sockets should be used for all ICs. The power should be off when inserting or removing ICs, and they should be protected from static discharge when taken out of their sockets.

We select the addresses of the ports during construction, by means of the jumpers which tie the individual lower-order address lines into the circuits. In the prototype, lines A8 and A9 select the output ports. The addresses thus are 8100 and 8200. Lines 12 and 13 select the input ports. The addresses thus are 9000 and A000.

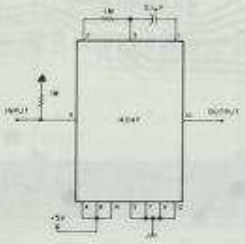


Fig. 8-4. Timer for I/O board.

Verifying Proper Operation

With the control panel in place to provide +5 volt power, the I/O board can be checked out. Before turning on the power though, it's well to check the +5 volt line on the board in order to ensure that a short to ground doesn't exist.

If all is well, we can check out the timer. This is most easily done by monitoring its output with a logic probe. Its output should be low, initially. If we momentarily ground the input of the timer, its output should go high for a fraction of a second and then go low again.

The control circuit on the I/O board should be checked out next. Using the debugging breadboard, we program the address lines, the R/W line and the Ø2 line to the states which will provide a control signal to each port, individually.

For example, if we want to check out the first output port (assuming that the address is 8100), we program A15 high, A14 low, and A8 high. If we then take the Ø2 line high momentarily, the control output to the port should go low momentarily.

If we want to check out the first input port, we do the same sort of thing, except that the R/W line (taken high) is involved rather than the Ø2 line. Of course, the corresponding control output goes high rather than low.

Once the control circuit has been tested, we can check out the ports themselves.

The easiest way to check out an input port is to turn the solid-state switches on by properly programming the address and R/W lines, using the debugging breadboard. Then, as a signal is applied to each bit, the corresponding line of the data bus should track that signal. This is the case since the latches in the port are programmed (by means of pull-down resistors) to track, rather than store the applied signal. If a high level is applied to the clock input of the latches, then they should hold the current data until the clock input is taken (or allowed to go) low again.

To check out an output port, we use a similar procedure. We program the appropriate address lines to the proper levels, using the debugging breadboard. We then program the data lines to any arbitrary states. If we then take the Ø2 line high momentarily, and then low again, the number on the data bus should be stored in the output port and be accessable at the output terminals.

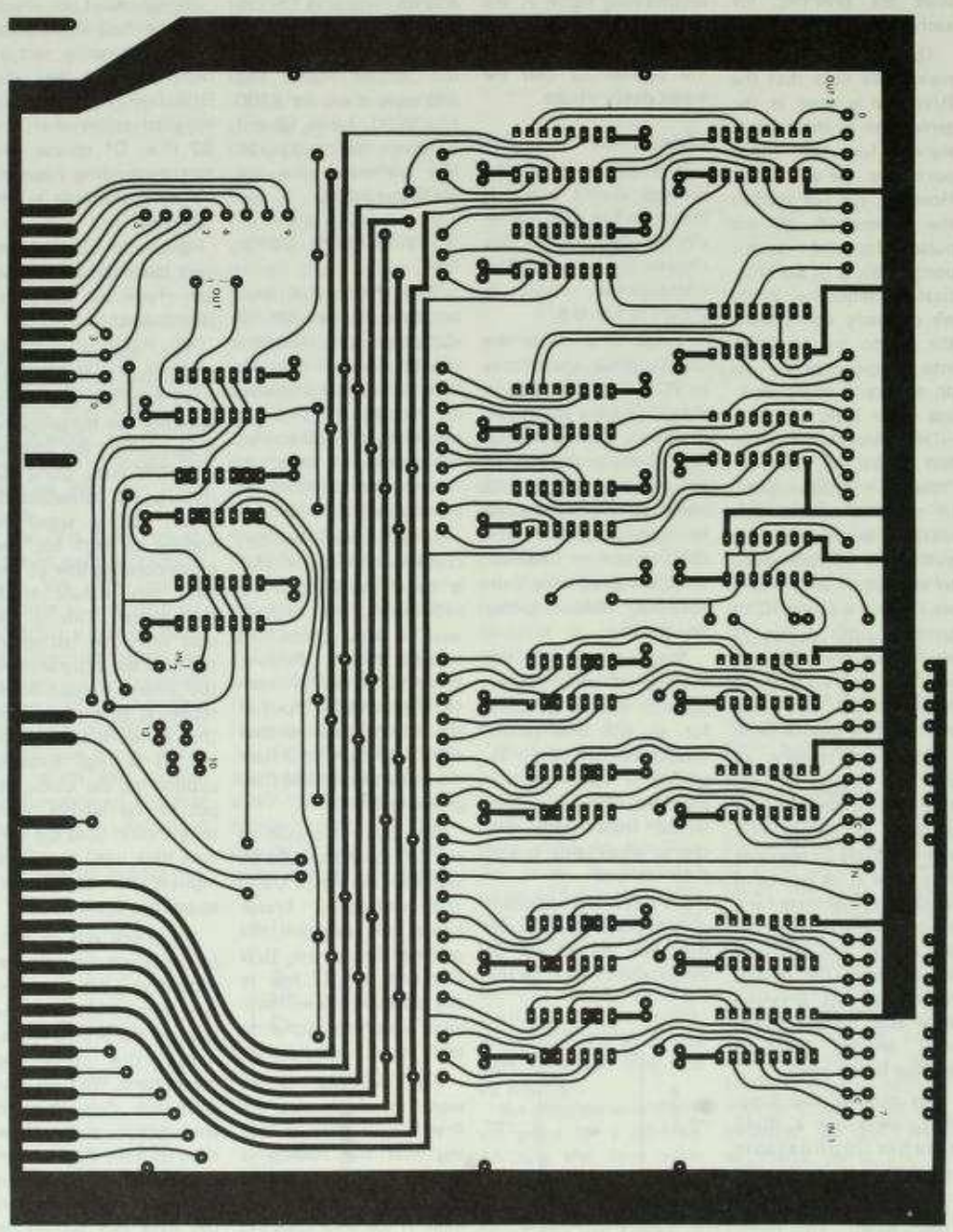


Fig. 8-5. Foil side of I/O board.

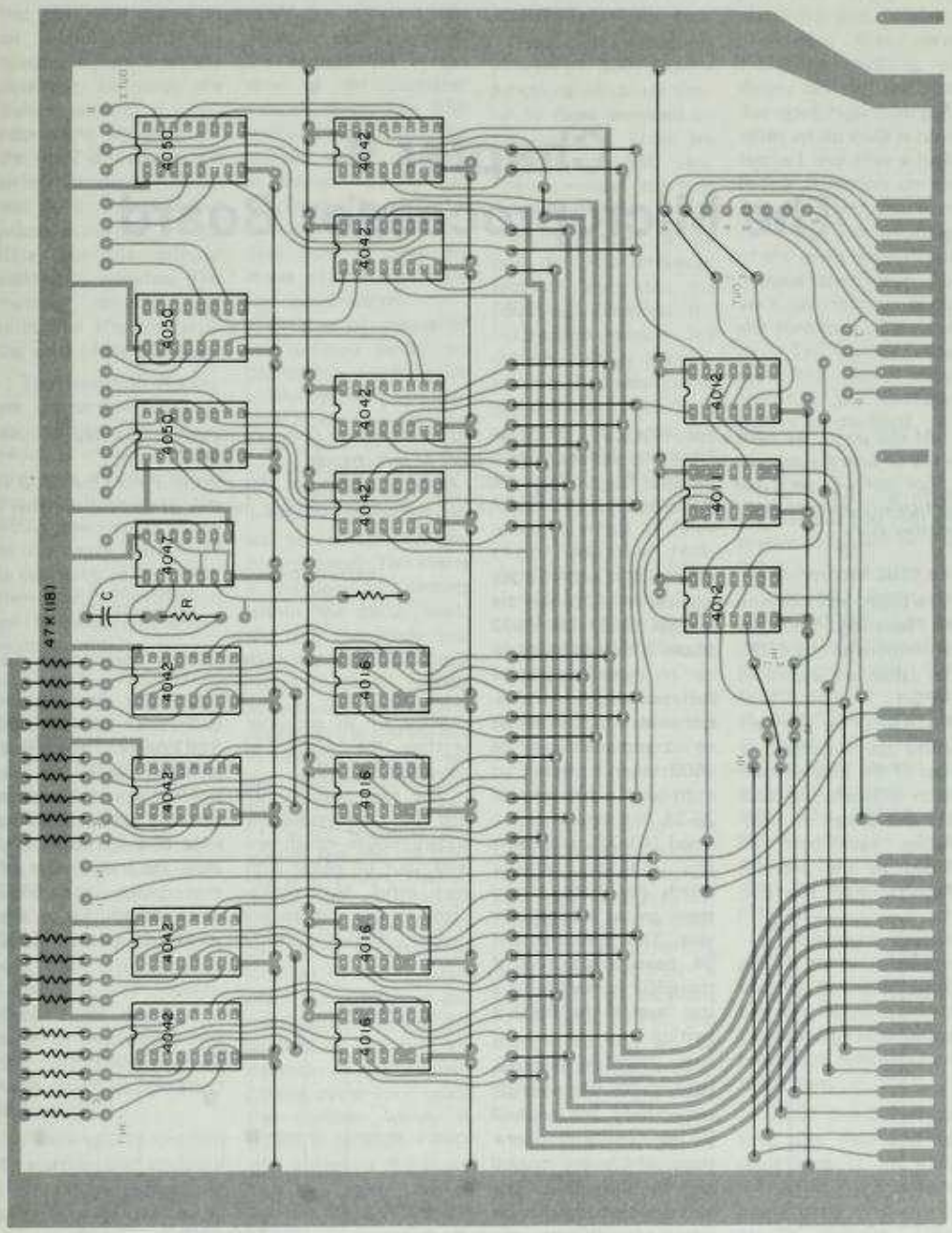


Fig. 8-6. Component side of I/O board. R and C are timer parts.

Chapter 9
The Microprocessor Board

At this point, we have a control panel, a DIPROM, and an I/O board. Only one module of our basic system – the uP board – remains to be built, and that's the subject of this chapter.

For our uP, we'll use the MOS Technology 6502. A major reason for this choice is that the 6502 is particularly easy to get "up and running." Of course, it's a good device on a number of counts, but that's another story.

The 6502 Microprocessor

To begin, let's examine the functions and pin connections of the 6502. The latter are shown in Fig. 9-1.

Actually, we can recognize the functions of most of the pins without much difficulty, because bits and pieces of information have been included in the previous chapters. For example, pins 9-20 and 22-25, labeled AO-A15, are the address lines. The 6502 places a sixteen-bit number on these lines, in effect pointing to a location with which it wishes to communicate. The 6502 then places an eight-bit number on pins 26-33, the data lines (labeled DO-D7) or accepts the eight-bit number which has been placed there by an external device. The R/W line, pin 34, bears a high level if the 6502 is reading or a low level if the 6502 is writing.

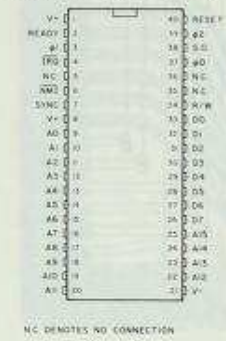


Fig. 9-1. Pin connections of the 6502.

Pins 37, 3, and 39, labeled Ø0, Ø1, and Ø2, respectively, are involved in the timing of operations which are carried out by the 6502. The pins are tied together by means of an RC-network. The constants of the network determine the frequency of an oscillator within the 6502, as shown in Fig. 9-2.

FREQUENCY, MHz	R	C
0.2	100K	10 pF
0.5	42K	10 pF
1.0	17K	10 pF

Fig. 9-2. Frequency determining components for 6502.

The two outputs, Ø1 and Ø2, are essentially square waves which are 180° out of phase with each other. Most operations in our system are properly carried out only if they are synchronized to Ø1 or Ø2. This is illustrated in Fig. 9-3.

The start of a cycle is signalled by the rising edge of a Ø1 pulse. Just after the rising edge of that pulse, the 6502 places an address on the address bus. If the 6502

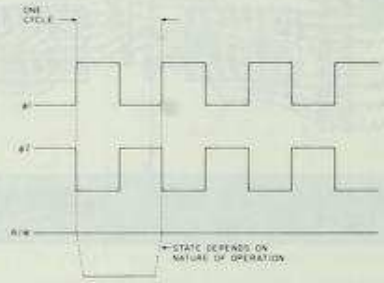


Fig. 9-3. Timing diagram of 6502.

is involved in read operations, it holds the R/W line high, and stores (in an internal latch) the number which is on the data bus, just after the trailing edge of the corresponding Ø2 pulse. If the 6502 is involved in a write operation, it takes the R/W line low and places a number on the data bus via internal solid-state switches. The number remains valid until just after the trailing edge of the Ø2 pulse.

The execution of a single instruction, such as the addition of a number which is stored in memory to a number which has been loaded into the 6502, may require as few as two cycles. If the 6502 is operating at 500 kHz, then such an instruction will be carried out in four microseconds.

If we understand what's been said about the address and data buses and the R/W, Ø1, and Ø2 lines, then we understand most of what we need about the way the 6502 operates once it's running. Let's look now at how we start it and stop it, and how we know at what address it will start. The latter is important, of course, since when we load a series of instructions into the DIPROM, the uP must start at the first instruction.

As we saw in Chapter 6, starting and stopping the 6502 is accomplished by means of the ready line, pin 2. If we pull the ready line high, the 6502 starts, and continues running until we pull the ready line low. When the ready line is pulled low, whatever address was on the address bus at the time of the transition remains there. The R/W line goes high, and the Ø1 and Ø2 signals continue as though nothing had happened. The 6502 disconnects itself from the data bus, so that the states of the data lines are indeterminant.

Fine – by means of the run-stop switch on the control panel, we've pulled the ready line low. Now how do we start the 6502 again, and at the proper address? Well, the first thing we do is momentarily depress the reset pushbutton on the control panel. This resets a number of circuits within the 6502. Next, we take the ready line high. During the next 5 cycles (of the Ø1, and Ø2 pulse trains) numbers will appear on the address bus in apparent random order. However, during the sixth cycle, the number FFFC will appear on the address bus and the R/W line will be high. That is, during the sixth cycle, the 6502 reads the number which is stored in location FFFC. It interprets the number as the less-significant half of the address of the first instruction in our program During cycle 7, it reads the number which is stored in location FFFD and interprets the number as the more-significant half of the address. During cycle 8, the 6502 loads the number which is stored at the address and interprets it as part of the first instruction of our program.

Two of the remaining pins on the 6502 provide functions which are similar to those provided by the reset pin. These are the IRQ and NMI pins. They provide interrupt functions.

For example, if we pull the IRQ (interrupt request) line low, the 6502 will complete the instruction which it's carrying out. It then interrupts the execution of the program which is in progress and reads the numbers which are stored at locations FFFE and FFFF. It interprets the numbers which it finds there as the less-significant and more-significant halves, respectively, of the address of the next instruction to be executed.

For some purposes, it's convenient to disable this function. We can do so by means of a particular instruction.

The function of the NMI (non-maskable interrupt) line is similar, except that a high-to-low transition on the NMI line causes the interrupt, the desired address is contained in locations FFFA and FFFB, and there is no way we can program the 6502 to ignore a signal on the NMI line.

In our basic system we won't use the interrupts. However, the NMI line may be useful after we expand the system to include a keyboard.

We'll have no need for the functions which are performed by the two remaining pins. Without pretending that we're explaining anything, we simply say that the Sync line goes high during Ø1 when an op code is being fetched and that a high-to-low transition on the S.O. line sets the overflow flag. The meanings of the terms involved will become clear later, but we'll say no more about the functions of the two pins. The curious reader can find more about them in the hardware manual on the 6502.

The Circuit

The circuit by which the 6502 is tied into the system is shown in Fig. 9-4.

The 6502 operates at about 200 kHz when the resistor and capacitor shown are used. The system can be operated at higher frequencies (up to about 1 MHz) depending on what devices are added to the basic system. The highest frequency can be determined by experiment after the system is completed.

All the input lines are pulled to one level or another in order to protect the inputs from static discharge if the other boards are removed from the system.

If the builder so desires, the data and address lines of the 6502 may be tied to the appropriate buses by means of 1k-10k resistors. These serve to limit the current which can flow if an ex-

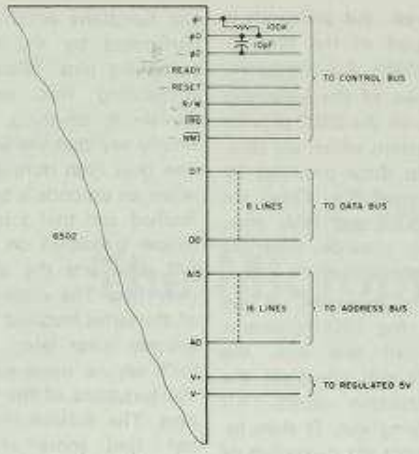


Fig. 9-4. Connections to 6502.

ternal device is improperly connected to one or the other of the buses. The resistors do tend to slow the system, though. In the prototype, the address pins are connected directly to the address bus, but the data pins are connected to the data bus via 1k resistors.

Construction

The foil side of the PC board and the component layout are shown in Figs. 9-5 and 9-6, respectively. The observant reader will quickly notice that the PC board is much too complicated to contain just the uP. We'll postpone a discussion of the remaining circuit until Chapter 15, however.

Those who choose one or the other alternatives to PC board construction shouldn't have any major problems, but the alternatives weren't investigated in the design of the prototype.

Verifying Proper Operation

To completely check out the 6502 would mean that we'd have to have the entire system operating. In the next chapter we'll do just that, but first there are some preliminary checks to be made.

We insert the control panel and uP board into sockets on the backplane and insert ICs CP5, CP6, CP7, and CP8 into their sockets on the control panel (power Off). Before we insert the 6502 into its socket, we turn the power on and verify again that the reset and run-stop switches have the expected effects on the two lines. We also verify that each remaining pin on the socket of the 6502 is at the expected logic level. The IRQ and NMI lines should be high. Pin 8 should be high also, since it is the +5 volt-power pin on the 6502. Pins 1 and 21 should be low since they're grounded. Finally, using an ohmmeter, we check to see that the timing resistor is properly connected.

Once we've done all this, we TURN OFF THE POWER, insert the 6502 into its socket and turn on the power again. With a logic probe or oscilloscope, we then check that the 6502 is operating. Regardless of the state of the run-stop switch, we should see something which approximates a square wave at the Ø1 and Ø2 pins. If the run-stop switch is thrown to *run*, the address lines should be changing states, more or less in time with Ø1 and Ø2, or a sub-multiple of either. A series of positive-going pulses should appear at the Sync pin.

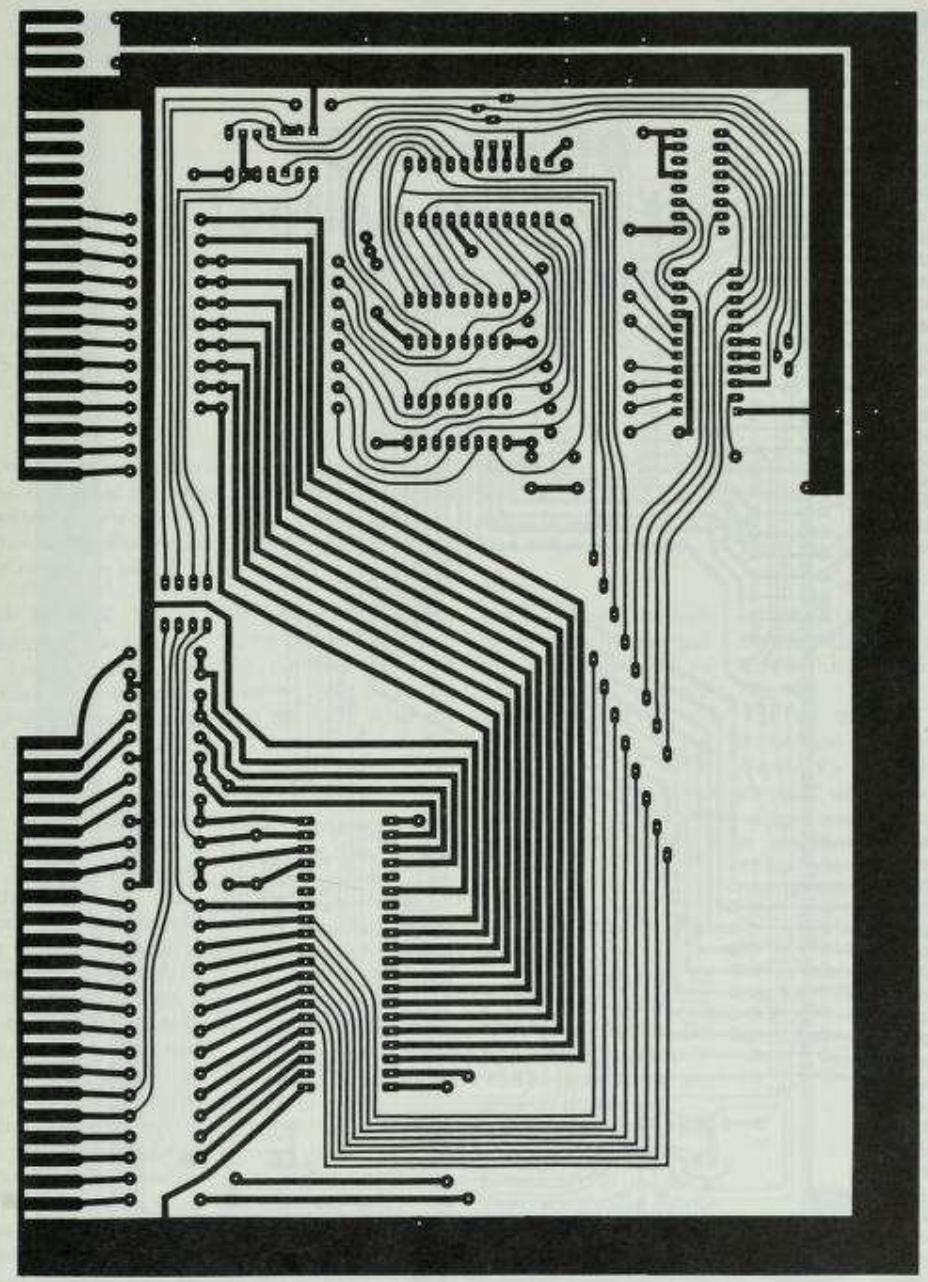


Fig. 9-5. Foil side of the uP board.

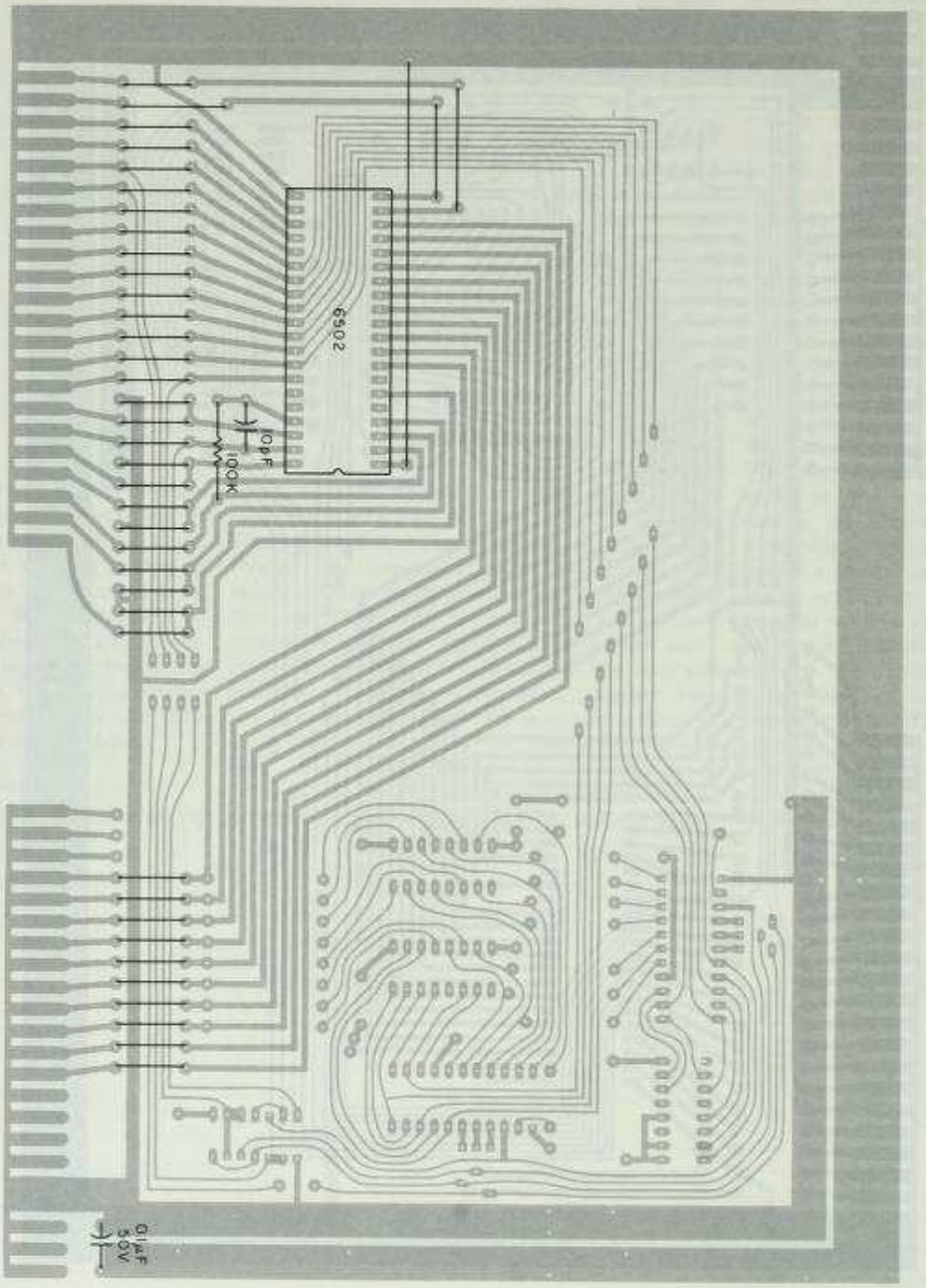


Fig. 9-6. Component side of the uP board.

Chapter 10
Testing the System with Software

Until now, we've been concerned almost entirely with hardware – ICs, PC boards, and the like. However, we've completed our basic system and tested the separate modules. At this point we must turn our attention to software – the sets of instructions which enable our system to perform useful tasks.

Our goal in this chapter is simply to learn enough about software to be able to check the system, and to have a little fun with it.

Inside The 6502

Before we can program the system intelligently, we have to understand a little about the internal workings of the 6502. For example, our first program will make use of two registers within the 6502. The first of these is the *accumulator* register (ACR). When the 6502 reads a number from the data bus, that number is deposited in the ACR. When the 6502 deposits a number on the data bus, that number originates in the ACR. The ACR, in effect, is an "I/O port" within the 6502. In this case the "outside world" is the data bus, however.

The ACR differs from a conventional I/O port in more ways than one, though, since it's also the register in which many manipulations of numbers originate. For example, before the 6502 performs an addition, one of the numbers which is involved must be loaded into the ACR. After the addition is carried out, the result remains in the ACR.

Our first program will also make use of the *program counter* (PC). The PC is a sixteen-bit binary counter which keeps track of the address that the 6502 places on the address bus. In fact, for the present, we can consider that the address bus is tied to the outputs of the PC.

The PC may be controlled either by the 6502 directly or though the program which the 6502 executes. For example, regardless of anything that's contained in a program, the 6502 places the number FFFC on the address bus (via the PC) during the sixth cycle after the reset button is activated. On the other hand, if we program locations FFFC and FFFD to contain, say, F000, then F000 will be placed on the address bus during the eighth cycle after the reset button is activated.

Introduction To Programs

With that preface, we can get down to the task of writing programs. The first step is to consider what a program is and what it's made up of.

Fundamentally, a program is nothing more than a set of numbers. They are chosen on the basis of rules which are provided by the manufacturer of the uP, and according to what it is that we want to accomplish. For example, we might store in the DIPROM the set of numbers which is shown in Fig. 10-1.

If we did and then started the system after pressing the reset button, the 6502 would read the numbers which we stored in locations FFFC and FFFD and place them (F0,FF) on the address bus. Next, the 6502 would read A9, the number which we stored in location FFF0. To a

LOCATION	CONTENTS
FFF0	A9
FFF1	42
FFFC	F0
FFFD	FF

Fig. 10-1. Machine code for 6502.

6502, the number A9 (under certain conditions) means "load the next consecutive number into the ACR." Because of this, the 6502 would then load the number 42 into the ACR. What would then happen depends on the number which is stored in location FFF2.

Before we go any farther, we should understand that we have no way of knowing that A9 means what it means to a 6502 unless the manufacturer tells us. For this purpose, we need a table of *operation codes* (op codes). An op code is a necessary ingredient in any instruction since it indicates what particular operation is to be carried out. An instruction may also contain an additional number or two. They tell us something about the number which is to be operated on (its address or its value, depending on the op code). That number is called the *operand*.

At this point, we should understand that storing the number A9 in locations FFF0 and FFF1 in the previous example would cause the 6502 to load A9 into the ACR. That is, the same number may mean different things at different times depending upon what the 6502 expects it to mean (op code or operand).

LOCATION	CONTENTS
FFF0	– –
1	A9
2	42
3	8D
4	00
5	82
FFFC	F1
D	FF

Fig. 10-2. Our first program.

NUMBER OF TIMES SINGLE-CYCLE BUTTON IS PRESSED	NUMBER ON ADDRESS BUS	NUMBER ON DATA BUS
6	FFFC	F1
7	FFFD	FF
8	FFF1	A9
9	FFF2	42
10	FFF3	8D
11	FFF6	?

Fig. 10-3. Results of single-cycling.

Our First Program

For our first effort, we'll write a very simple program – one which will cause the 6502 to load a number from memory into the ACR and then move the number from the ACR into an output port. Since we can specify the number and then examine the port, we can see if the program is executed properly. If it is, then our system is working in at least a modest way.

To load the number from memory, we'll use the instructions which we discussed earlier. To store the number in the output port requires a three-word instruction. The op code is 8D. It means "store the contents of the ACR in the location which is specified by the next two words, taking those two words to be the eight less-significant and eight more-significant bits of the address, in that order".

If the address of the output port is 8200, if we want to store, say, the number 42, and if we want the program to start at, say, location FFF1, then our program is as shown in Fig. 10-2.

After we've loaded the program into the DIPROM, we should check to see that each location contains what we think it does. The easiest way to do this is by using the debugging breadboard. AFTER THE uP BOARD HAS BEEN REMOVED FROM THE SYSTEM (power off, etc.), we tie the R/W line high, program the address lines through the sequence of locations which we want to examine, and verify that the correct numbers appear on the data bus in the correct sequence. Assuming that all is in order, we then UNPROGRAM ALL LINES and replace the uP board (power off, etc.).

To run the program, we set the run-stop switch to *stop* and turn on the power. Actuating the reset button and then pressing the single-cycle button six times should cause FFFC to appear on the address bus and F1 to appear on the data bus. As we continue pressing the single-cycle button, the program is executed. The sequence is summarized in Fig. 10-3.

We can work our way though the first part of the program a single cycle at a time. However, it turns out that the 6502 can't be "single-cycled" through a WRITE operation. Once such an operation is started, the entire instruction is executed at full speed.

A complication arises if we continue to actuate the single-cycle button after the program is executed. The 6502 expects to find more instructions, but there are none. If we want to run the entire program at full speed, we must provide some way to stop the 6502 from running wild after it has carried out the final instruction. The 6502 doesn't have a HALT instruction, so we can't use that approach. What we can do is provide an endless loop of instructions for the 6502 to execute. This is done in the program which is shown in Fig. 10-4.

We've added four words to our program. The first of these, EA, means "no operation" (NOP). When the 6502 encounters EA in a program sequence, it executes the instruction. But the execution causes nothing of consequense as far as the outside world is concerned, except a delay of two cycles. On our

LOCATION	CONTENTS
FFF0	– –
1	A9
2	42
3	8D
4	00
5	82
6	EA
7	4C
8	F6
9	FF
A	– –
B	– –
C	F1
D	FF

Fig. 10-4. An improved program.

program, it's the start of the endless loop. The next instruction, 4C, means "jump to the address specified by the next two consecutive words, taking those two words to be the eight less-significant and the eight more-significant bits of the address, respectively." In the execution of this instruction, the 6502 loads the next two words and stores them in the program counter. The next instruction which the 6502 executes is the instruction which is located at that address. In our program, that's the address just before the JUMP instruction. It contains the NOP instruction. The 6502 executes the NOP instruction and then, again, the JUMP instruction. This process continues until we intervene manually.

If we step through the program, or run it at full speed, the result is the same. The number which is stored at location FFF2 is written into the output port at location 8200. The 6502 then enters the endless loop of instructions.

If we change one instruction in the program, we can verify that an input port works properly. We simply cause the 6502 to load a word from the input port rather than from the DIPROM. To do this, we use the instruction AD. The 6502 interprets AD as "load the ACR with the contents of the location specified by the next two words, taking those words to be the eight less-significant and eight more-significant bits of the address respectively." If the address of the input port is 9000, the program which is shown in Fig. 10-5 will move the contents of the input port into the output port.

LOCATION	CONTENTS
FFF0	AD
1	00
2	90
3	8D
4	00
5	82
6	EA
7	4C
8	F6
9	FF
A	— —
B	— —
C	F1
D	FF

Fig. 10-5. Program to test an input port.

At this point, we have our basic system up and running.

Chapter 11
Programming I

Until now, we've been more or less pulling instructions out of the air, in order to write a simple program or two. Before we write many more, it's useful to systematically examine a number of instructions. While we're at it, we'll work up a set of abbreviations for the instructions. We'll later find that as our programs become longer, an easy way to write them is to use the abbreviations and then translate the programs into numbers.

Our primary goal in this chapter, then, is to become more systematic in our handling of software. Before we begin, though, there are two topics which we have to discuss. The first concerns a register within the 6502.

The Status Register

The register is the *status register* (SR). It's a collection of flip-flops, each of which can be set to 1 or reset to 0 by the 6502. Six are of interest to us. The state of each flip-flop depends on what happens when particular instructions are carried out. The flip-flops are often called *flags* since they are indicators, as we'll see in a moment.

One of the flip-flops in the SR is called the *zero-result* flip-flop (*Z* flip-flop, or *Z* flag). If 0 is loaded into the ACR or left there as a result of some operation, the Z flag is set to 1. If other than 0 is loaded into the ACR or left there as a result of some operation, the Z flag is reset to 0.

Another flag in the SR is the *negative-result* (N) flag. If a negative result is loaded into the ACR or left there as the result of some operation, the N flag is set to 1, while a positive result causes it to be reset to 0.

Each of these flags can be affected in other ways as we'll see later. We'll also see how we can make use of the flags.

Methods of Addressing

The second topic which we have to discuss concerns the way in which we specify the address of an operand.

As we saw in the previous chapter, we can load numbers into the ACR by two different means. The result is the same in either case — a number is loaded — but the locations of the numbers are specified in different ways.

In one case, the entire sixteen-bit address follows the op code, making up a three-word instruction. Because the entire address of the operand is completely specified, we say that such instructions involve *absolute addressing*.

In the second case, the eight-bit operand itself follows the op-code, making up a two-word instruction. Because the operand immediately follows the op-code, we say that such instructions involve immediate addressing.

As we consider more instructions, we'll see that many of them can accomodate both of these methods. In future chapters, we'll encounter additional methods, as well. With that, we're ready to examine instructions.

A Catalog of Instructions

A complete listing of all the instructions which the 6502 recognizes is shown in Appendix 1. The purpose of each instruction, its op codes and a list of the flags which it affects are included. At this point, much of the listing may be unintelligible, but by the end of the book that won't be so.

We should recognize at least a few of the instructions. For example, we've already used the one which loads a number into the ACR (LDA) and the one which stores the number that is contained in the ACR (STA). Absolute addressing can be used with either, but immediate addressing can't be used with the STA instruction. LDA affects the N and Z flags. STA affects none.

We've used two other instructions as well. The first is the NO OPERATION (NOP) instruction. Its major effect is simply to cause a slight delay in the execution of a program.

The second is the JUMP (JMP) instruction. The operand of a JMP instruction is used by the 6502 as the address for the next instruction which it fetches. Immediate addressing can't be used (why would we want to?) and the instruction affects no flags.

The next set of instructions to consider contains those which implement logical operations. Three are available — and (AND), or (ORA), and exclusive or (EOR). Each affects the N and Z flags. Immediate and absolute addressing can be used with each.

A similar instruction is the add (ADC) instruction. In its execution, the operand is added to the contents of the ACR and the result is left in the ACR, as we'd expect. However, there's more to consider.

When we discussed the addition of binary numbers, we found that there can be a carry, just as there is in decimal addition. Within the SR of the 6502, there is a flag called the *carry* (C) flag which is set to 1 if a carry results from the execution of an ADC instruction. If no carry results, the C flag is reset to 0. For example, if the ACR contains 00110101, and 111001010 is added to it, the result is 11011010. No carry is generated, and the C flag is reset to 0. On the other hand if the ACR contains 10110101 and 11100101 is added to it, a carry is generated and the C flag is set to 1.

Actually, the execution of an ADC instruction does more than just add the two eight-bit numbers and generate a carry (if appropriate). In the process, the previous content of the C flag is added into the sum. For example, if the C flag has been set to 1 by a previous operation, if the ACR contains 00110101 and if 11100101 is added to it, the ACR will contain 11011011, not 11011010. Performing that addition, though, resets the C flag to 0, because no carry is generated in the execution. Because of all this, the instruction is more properly called the ADD WITH CARRY instruction. If that were the complete story, the situation would be nearly intolerable. Adding two numbers might produce either of two results, depending on whether the C flag had been set or reset as the result of some previous operation. Fortunately, there's more.

Instructions are available which allow us to set or clear (reset) the C flag as we choose. These are the SET CARRY (SEC) and CLEAR CARRY (CLC) instructions. Unless we're performing a multiple-precision addition (add lower eight bits of the numbers, generate carry if appropriate, add next higher eight bits of the numbers and the carry, etc.) we should always clear the C flag before performing an addition.

So far, so good, but there's still more to consider about the process of addition.

If the eight-bit numbers which are being added are unsigned binary numbers ($1111\ 1111_2$ represents 255_{10}, not -1_{10}), then the status of the C flag indicates whether or not the result of the addition exceeds the range which can be represented (0 to 255). On the other hand, if we're using signed binary numbers ($1111\ 1111_2$ represents -1_{10}), then the C flag has no significance. In effect, we're summing a pair of seven-bit numbers, since the eighth bits represent the signs of the numbers.

All this was taken into account in the design of the 6502. To understand how, we first must recognize that the eight-bit result of the addition of two eight-bit numbers is the same, regardless of the convention which we're using. We merely change the way in which we interpret the result.

What does depend on the convention which we're using is the indicator which tells us if the result is too large to represent with one word. As we've seen, if we treat the result as an unsigned number, the C flag is the indicator.

The SR contains another flag which is the indicator if we treat the result as a signed number. It's call the *overflow* (V) flag. It's simply a flag which indicates a carry from the seventh bits. Unlike the C flag, though, the V flag doesn't affect the result of an addition. It merely tells us something about that result. When the 6502 performs an addition, it sets or resets both the C and V flags to their appropriate values. We must choose which to use.

The CLEAR OVERFLOW (CLV) instruction

ACR	0000	0011	
MEMORY	0000	1100	
ACR	0000	1111	C = 0, V = 0
ACR	0000	0011	
MEMORY	0111	1110	
ACR	1000	0001	C = 0, V = 1
ACR	1111	1101	
MEMORY	1111	1111	
ACR	1111	1100	C = 1, V = 0
ACR	1000	0000	
MEMORY	1000	0000	
ACR	0000	0000	C = 1, V = 1

Fig. 11-1. Results of additions.

allows us to reset the V flag to 0.

Examples of additions which act in various ways on the C and V flags are shown in Fig. 11-1.

Just as there is an ADD instruction, there is also a SUBTRACT (with borrow, SBC) instruction. Unless we're performing a multiple-precision subtraction, we should always set the C flag to 1 before the subtraction. The 6502 performs the subtraction by the method of two's complements.

The C flag is set if the result of a subtraction is greater than or equal to 0. It's reset to 0 if the result is less than 0 (a borrow). The V flag is set if the result is outside the range +127 to -128, otherwise it is reset.

In addition to performing additions and subtractions on binary numbers, the 6502 also will carry out analogous operations on BCD numbers. If we set the decimal (D) flag in the SR by using the SED instruction, then in any arithmetic operations which are carried out, the eight-bit numbers will be interpreted as a pair of BCD numbers. In BCD addition, of course, a carry is generated if the result exceeds 99_{10} ($1001\ 1001_2$). We exit from the BCD mode by using the CLEAR DECIMAL MODE (CLD) instruction.

Both immediate and absolute addressing can be used with the ADC and SBC instructions.

The next instructions which we'll examine operate on the contents of the ACR and a memory location. In that respect they're similar to the instructions which we've already examined. However, unlike those instructions, these don't change the contents of the ACR. Rather, they merely set or reset flags in the SR, depending on the results of the operation.

The first of these is the COMPARE (CMP) instruction. Executing this instruction subtracts the operand from the contents of the ACR. The Z flag is set if the result is 0, otherwise the Z flag is reset. The N flag is set or reset, depending on bit 7 of the result, in the way which we'd expect. The C flag is set if the operand is less than or equal to the contents of the ACR, otherwise the C flag is reset. Either immediate or absolute addressing can be used.

The second of these instructions, the BIT TEST (BIT) instruction, is similar except that a logical rather than an arithmetic operation is involved. Execution of this instruction ANDs the operand with the contents of the ACR. The Z flag is set if the result is 0, otherwise it is reset. The N flag and V flag are set to the values of bit 7 and bit 6 of the original operand, respectively. Immediate addressing is not available.

The next instructions which we'll consider operate on a number, moving all the individual

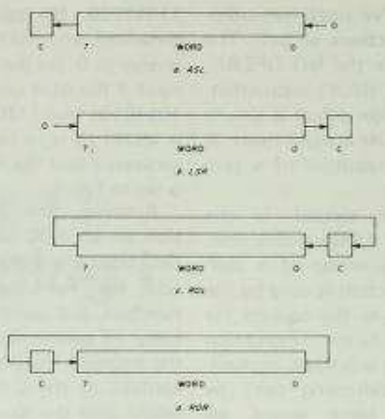


Fig. 11-2. Effects of shift and rotate instructions.

bits without changing their relative positions. These are the SHIFT and the ROTATE instructions.

For example, the SHIFT LEFT (ASL) instruction moves each bit in the ACR left by one position. The most significant bit is shifted into the C flag and an 0 is placed in the least-significant bit, as shown in Fig. 11-2(a). If we clear the C flag, load 1010 1010 into the ACR, and execute the ASL instruction, the ACR then contains 0101 0100, and the C flag is set to 1.

In this example, the operand of the ASL instruction is located in the ACR and the instruction is only one word long. We refer to this as *accumulator addressing*, since the "address" of the operand is the ACR.

The ASL instruction affects the N, Z, and C flags, setting or resetting each depending on the results of the shift.

A SHIFT RIGHT instruction (LSR) is also available. It causes the manipulation which is shown in Fig. 11-2(b). The LSR instruction affects the N, Z, and C flags. It always resets the N flag since a 0 is shifted into the sign bit.

If we want to circulate the bits in a number but not lose any of them, we use a ROTATE instruction. For example, the ROTATE LEFT (ROL) instruction causes the manipulation which is shown in Fig. 11-2(c).

A ROTATE RIGHT (ROR) causes the manipulation which is shown in Fig. 11-2(d).

The ROTATE instructions affect the N, Z and C flags.

Most of the instructions which we've examined to this point affect one or more of the flags in the SR. The last major set of instructions which we'll discuss in this chapter is comprised of the so-called BRANCH instructions. By means of these instructions, we can cause the system to examine the results of a

previous operation (by means of the flags in the SR) and then carry out one of two pre-determined operations depending on those results.

Each of the BRANCH instructions causes the 6502 to branch (jump) to a specified address depending on whether one of four flags in the SR has been set or reset. Whether the 6502 returns to the original sequence of instructions depends entirely on the instructions which start at the location to which the branch occurs.

A typical BRANCH instruction is the BRANCH ON CARRY CLEAR (BCC) instruction. In its execution, the C flag is examined. If it's a 0, the branch occurs. If it's a 1, the branch does not occur and the remainder of the program is carried out as if no BRANCH instruction were present.

The instruction consists of two words. The first is the op code for the instruction. The second is a pointer — a number which is added to the contents of the PCR, indicating to where the branch should be made. The effect of the instruction is illustrated in Fig. 11-3.

As the program is being executed a branch will not occur if the C flag is 1. The instruction at location FFF4 will be executed. However, if the C flag is 0, the branch will occur and the instruction at location FFF7 will be executed.

LOCATION	INSTRUCTION/CONTENTS
FFF2	BCC
3	03
4	ASL
5	ASL
6	ASL
7	LDA

Fig. 11-3. Effects of branch instruction.

The branch isn't made to location FFF6, because as soon as the 6502 loads the +3 from location FFF3, the PC is incremented to FFF4 in anticipation of fetching the contents of that location. Thus, the +3 is added to FFF4 rather than FFF3.

The 6502 treats the pointer as a signed number so that branches can be made forward or backward.

Since the pointer of a BRANCH instruction is not interpreted as an operand but rather as part of an address, we can't properly apply the *immediate addressing*. Rather, since the pointer indicates a location relative to the location of the branch instruction, we describe this as *relative* addressing.

Chapter 12
The Random-Access Memory

In part I of this book, we built a small system and learned a little about programming it. To this point, though, we have not put it to practical use. In this, the second part of the book, we'll change that. We'll put the system to good use in several applications in the ham shack. Along the way, we'll add more hardware and learn more about the system and how to program it.

The first hardware which we'll add is the subject of this chapter. It's a small, random access memory *(RAM). The name isn't all that helpful in understanding what the device does. It's more properly called a* read-write *memory, since the uP can write numbers into it as well as read numbers which have been written into it. However, we'll continue to refer to it as a RAM because that's the common practice.*

The Need For RAM

The RAM serves two important functions in our system. First, it provides a scratchpad for the uP. That is, the uP can store information in the RAM and recall it, as necessary. Of course, we could accomplish the same thing in the basic system by tying the outputs of an output port to the inputs of an input port. However, the ports then would be unavailable for other purposes, and the method would be quite expensive if we had to store more than three or four numbers simultaneously.

The second function which the RAM serves is to provide temporary storage for programs. When we exceed the storage capacity of the DIPROM, we can write the program into the RAM. We'll see how that's done in a later chapter.

The RAM IC

At the time this is written, the most straightforward way to construct a RAM is to use MOS ICs which are available for the purpose. Other completely different components which can be used as storage elements are available, but they are neither as inexpensive nor as convenient to use.

Typical RAM ICs, which are presently available to the hobbyist, contain 1024 one-bit locations. The way in which the bits are organized varies from one IC to another. For example, we'll use a pair of 2101 ICs, each of which will hold 256 four-bit numbers. The memory, then, will hold 256 eight-bit numbers. For some purposes this would be a very small memory, but it will serve our immediate needs very nicely. Later, we'll see how it can be expanded.

Fig. 12-1. Pin configuration of the 2101.

The inputs and outputs of the 2101 are TTL-compatable, which means that the device will work properly with the 6502. As long as the power is not turned off, each location will retain the number which we store there until we store another number. That is, information is read out non-destructively.

The pin configuration of the 2101 is shown in Fig. 12-1.

We can divide the pins into five groups. The first contains pins 8 and 22, the power connections.

An eight-bit address is applied to the second group, pins 1-7 and 21 (the *address* pins). If a number is to be written

into the memory, it's applied to the third group, pins 9, 11, 13, and 15 (the data input pins). If a number is to be read from the memory, it appears at pins 10, 12, 14, and 16 (the data output pins).

The fifth group of pins includes those which are used to control the flow of data to and from the RAM. Two of these, pins 17 and 19, are called chip-enable (CE) pins. Unless a high level is applied to pin 17 and a low level is applied to pin 19, the chip is effectively disconnected from the circuit.

Pin 18 is called the output disable (OD) pin. When a high level is applied to this pin, solid-state switches within the IC are turned off, disconnecting the data-out pins from the memory. Conversely, a low level turns the switches on.

Pin 20 is called the read/write (R/W) pin. However, to avoid confusing it with the R/W line of our system, we'll call it the WP pin, where WP stands for write pulse. When we want to write a word into the RAM, we apply a negative-going pulse to the WP pin. Of course, other pins must have been taken to appropriate levels before the pulse is applied so that what we want to be written is written where we want it.

The details of a WRITE operation are conveniently summarized in the timing diagram which is shown in Fig. 12-2.

The first point to notice about the diagram is that after the applied address has become stable, we must wait a specified time before initiating the write pulse. In our system this causes no problem, since we can use the Ø2 pulse train as one of the inputs to the circuit which generates the pulse. The address which is generated by the uP is applied to the address bus at the beginning of a cycle, while the rising edge of the corresponding Ø2 pulse does not occur until midway in the cycle.

The second point to notice about the diagram is that the width of the write pulse must be at least a certain minimum value. This causes no problem. We can simply adjust the clock generator of the 6502 until a Ø2 pulse is as wide as required.

The third point to notice about the diagram is that the CE and data lines must be stable for a specified time before the trailing edge of the write pulse occurs. This causes no problem. If necessary, we can simply adjust the clock generator of the 6502 to accommodate the requirement.

The fourth point to notice about the diagram is that the address and data lines must be stable for specified times after the trailing edge of the write pulse. This likely will cause no problem, since the 6502 holds these lines stable for a time after the falling edge of the corresponding Ø2 pulse. We can minimize the possibility of a problem by introducing no unnecessary delay in the circuit which generates the write pulse. However, if the lines aren't stable long enough, we can prolong the time slightly by inserting buffers in the address and data lines. Because of propagation delays in the buffers, the lines at the memories will remain stable for an additional 50 nanoseconds or so. This is a two-edged sword, though. Any such delay will also retard the initial stabilization of the data and address lines so that we may have to run the system a little more slowly than otherwise would be the case.

A diagram which summarizes the timing that is involved in reading a number from the 2101 is shown in Fig. 12-3. The important point to notice is that a specified time must elapse between when the applied address becomes stable and when the number from the 2101 becomes valid. This is called the *access time* of the memory. It tells us how "fast" the memory is. While the access time of the 2101 is 1000 nanoseconds, another version (the 2101-1) has an access time of only 500 nanoseconds. Thus, if 2101-1s are used, the clock generator of the 6502 can be run about twice as fast as when 2101s are used.

The Interface

The circuit which we'll use in order to interface the 2101s to our system is shown in Fig. 12-4.

Three 4050 buffers are used to drive the address

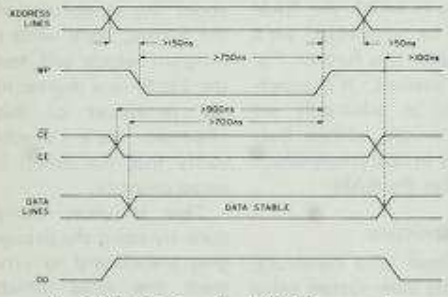


Fig. 12-2. Timing of a WRITE operation.

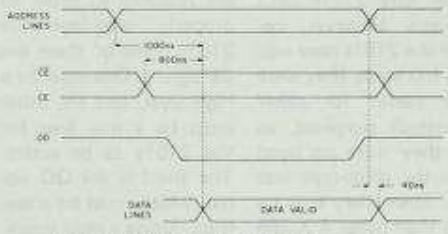


Fig. 12-3. Timing of a READ operation.

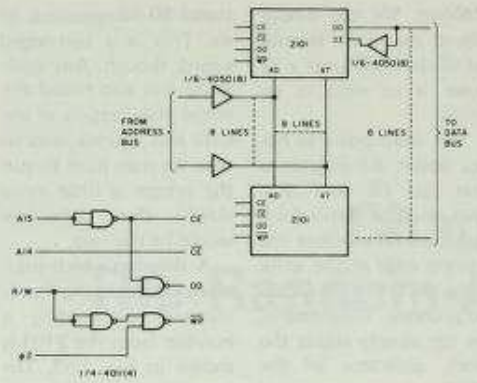


Fig. 12-4. Interface to RAM.

and data inputs of the 2101s. During breadboarding, the overall speed of the system apparently could be increased when some (but not all) 2101s were used, by including the buffers.

1k resistors are inserted in the data lines in order to protect the 2101s and the 6502 from each other in case of a malfunction.

At this point, the more advanced reader may wonder why 2101s are used rather than, say, 2111s. The two are similar except that in the latter the input and output lines are tied together within the IC so that fewer connections are required, and the ICs are physically smaller.

In fact, there is no reason why 2111s can't be used. However, because the 2101s have separate I/O leads, they were more useful for other (unrelated) purposes, so that they were on hand when the prototype was built. Since they cost no more than other 4 X 256 ICs, 2111s included, there was no reason not to use them.

The assignments of the address and data lines were chosen on the basis of convenience in laying out the PC board. It makes no difference whether we use the same labels as does the manufacturer, as long as we're consistent. If we apply, say, 1010 0101 to the data bus and 0000 1111 to the address bus and then apply a write pulse to the RAM, a particular pattern of bits will be stored in a particular set of locations. If we later apply 0000 1111 to the address bus, then 1010 0101 will again appear on the data bus and the bits will be in the proper sequence.

Four control signals are required in order to properly interface the 2101s. Two of these are CE signals. One must be a high level, and the other must be a low level for the 2101s to be active. The third is the OD signal, which must be a low level during a read operation. The fourth is a negative-going pulse, which is used in order to write a number into the 2101s.

Much earlier in the book, we adopted the convention that for addresses which involve the RAM, we'd use those in which both A14 and A15 are low. Taking that into account, we can formulate the control signals as:

$$\overline{CE} = \overline{A15}$$

$$CE = A14$$

$$OD = \overline{\overline{A15} \cdot R/W}$$

$$WP = \overline{\emptyset 2 \cdot \overline{R/W}}$$

A14 can be used directly as CE. The remaining three control signals are derived by IC RA6, a quad two-input NAND gate.

Nothing has been said about address lines A8-A13, since they're not included in the decoding scheme. This creates a situation which is analagous to that which exists when a less-than-full-size DIPROM is used. That is, each location in the RAM can be addressed in more than one way, since we can place any combination of 1s and 0s on address lines A8-A13 with no effect. When we discuss the ways in which the RAM can be expanded, we'll consider this further. For the present, it's convenient to arbitrarily set the unused address lines to 0s in operations which involve the RAM.

Construction

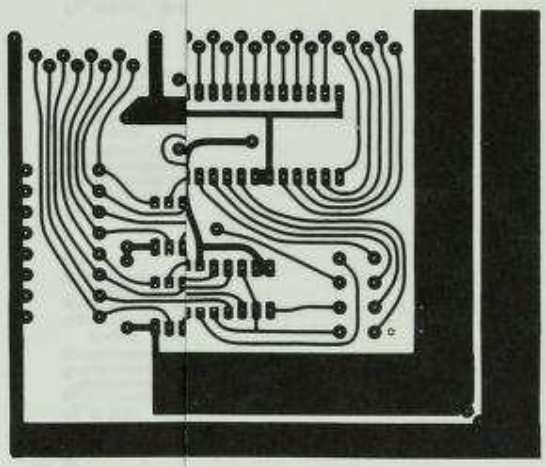
Those who constructed the basic system using the PC layouts which are included in this book have very little work to do in adding the RAM. It occupies the previously unused circuit on the uP board. The component layout is shown in Fig. 12-5.

Those who chose one of the alternatives to PC board construction should have no serious problems. All functions of the RAM were breadboarded before the prototype was constructed.

Verifying Proper Operation

With the exception of the 6502, all the ICs in the basic system provide fairly simple functions. If such an IC seems to be defective, we can check that all possible combinations of inputs produce the expected outputs. A moment's reflection will indicate that an exhaustive test of the RAM would be rather time-consuming. We would have to write and read each of 256 different words into and out of each of 256 different locations, and check that no unaddressed location had been affected by any operation. In the next chapter, after we've learned a little more about the capabilities of the system, we'll write a program which will test the 2101s to a degree. In the remainder of this chapter, we'll simply verify that the circuit is wired properly.

This is most easily done by using the debugging breadboard to program the lines which drive IC RA6 (6502 out



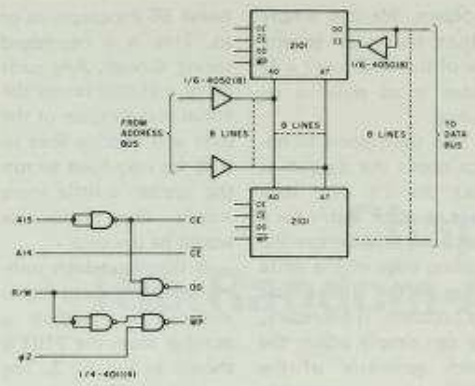


Fig. 12-4. Interface to RAM.

and data inputs of the 2101s. During breadboarding, the overall speed of the system apparently could be increased when some (but not all) 2101s were used, by including the buffers.

1k resistors are inserted in the data lines in order to protect the 2101s and the 6502 from each other in case of a malfunction.

At this point, the more advanced reader may wonder why 2101s are used rather than, say, 2111s. The two are similar except that in the latter the input and output lines are tied together within the IC so that fewer connections are required, and the ICs are physically smaller.

In fact, there is no reason why 2111s can't be used. However, because the 2101s have separate I/O leads, they were more useful for other (unrelated) purposes, so that they were on hand when the prototype was built. Since they cost no more than other 4 X 256 ICs, 2111s included, there was no reason not to use them.

The assignments of the address and data lines were chosen on the basis of convenience in laying out the PC board. It makes no difference whether we use the same labels as does the manufacturer, as long as we're consistent. If we apply, say, 1010 0101 to the data bus and 0000 1111 to the address bus and then apply a write pulse to the RAM, a particular pattern of bits will be stored in a particular set of locations. If we later apply 0000 1111 to the address bus, then 1010 0101 will again appear on the data bus and the bits will be in the proper sequence.

Four control signals are required in order to properly interface the 2101s. Two of these are CE signals. One must be a high level, and the other must be a low level for the 2101s to be active. The third is the OD signal, which must be a low level during a read operation. The fourth is a negative-going pulse, which is used in order to write a number into the 2101s.

Much earlier in the book, we adopted the convention that for addresses which involve the RAM, we'd use those in which both A14 and A15 are low. Taking that into account, we can formulate the control signals as:

$$CE = \overline{A15}$$
$$\overline{CE} = A14$$
$$OD = \overline{A15} \cdot R/W$$
$$WP = Ø2 \cdot \overline{R/W}$$

A14 can be used directly as CE. The remaining three control signals are derived by IC RA6, a quad two-input NAND gate.

Nothing has been said about address lines A8-A13, since they're not included in the decoding scheme. This creates a situation which is analagous to that which exists when a less-than-full-size DIPROM is used. That is, each location in the RAM can be addressed in more than one way, since we can place any combination of 1s and 0s on address lines A8-A13 with no effect. When we discuss the ways in which the RAM can be expanded, we'll consider this further. For the present, it's convenient to arbitrarily set the unused address lines to 0s in operations which involve the RAM.

Construction

Those who constructed the basic system using the PC layouts which are included in this book have very little work to do in adding the RAM. It occupies the previously unused circuit on the uP board. The component layout is shown in Fig. 12-5.

Those who chose one of the alternatives to PC board construction should have no serious problems. All functions of the RAM were breadboarded before the prototype was constructed.

Verifying Proper Operation

With the exception of the 6502, all the ICs in the basic system provide fairly simple functions. If such an IC seems to be defective, we can check that all possible combinations of inputs produce the expected outputs. A moment's reflection will indicate that an exhaustive test of the RAM would be rather time-consuming. We would have to write and read each of 256 different words into and out of each of 256 different locations, and check that no unaddressed location had been affected by any operation. In the next chapter, after we've learned a little more about the capabilities of the system, we'll write a program which will test the 2101s to a degree. In the remainder of this chapter, we'll simply verify that the circuit is wired properly.

This is most easily done by using the debugging breadboard to program the lines which drive IC RA6 (6502 out

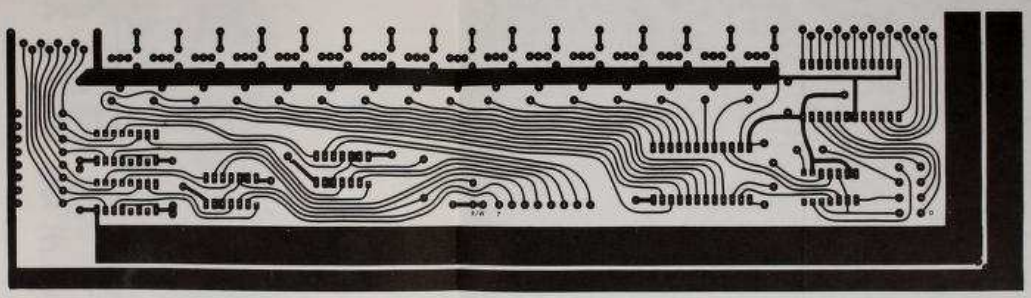
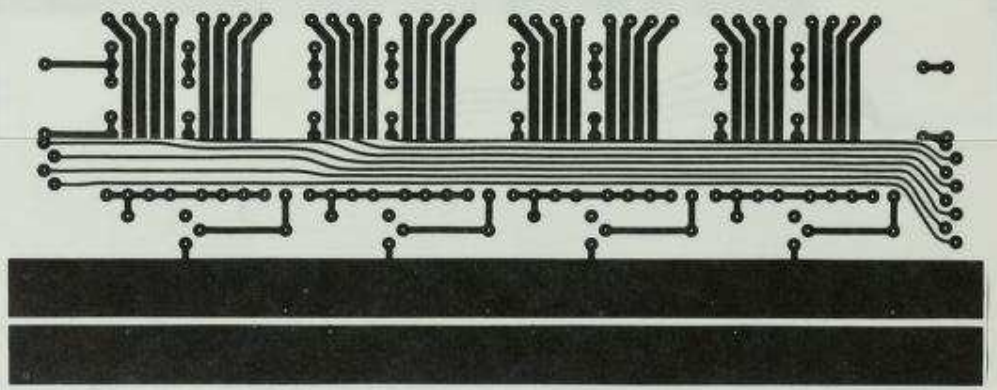


Fig. 7-8. Foil side of input/output board, DIPROM.



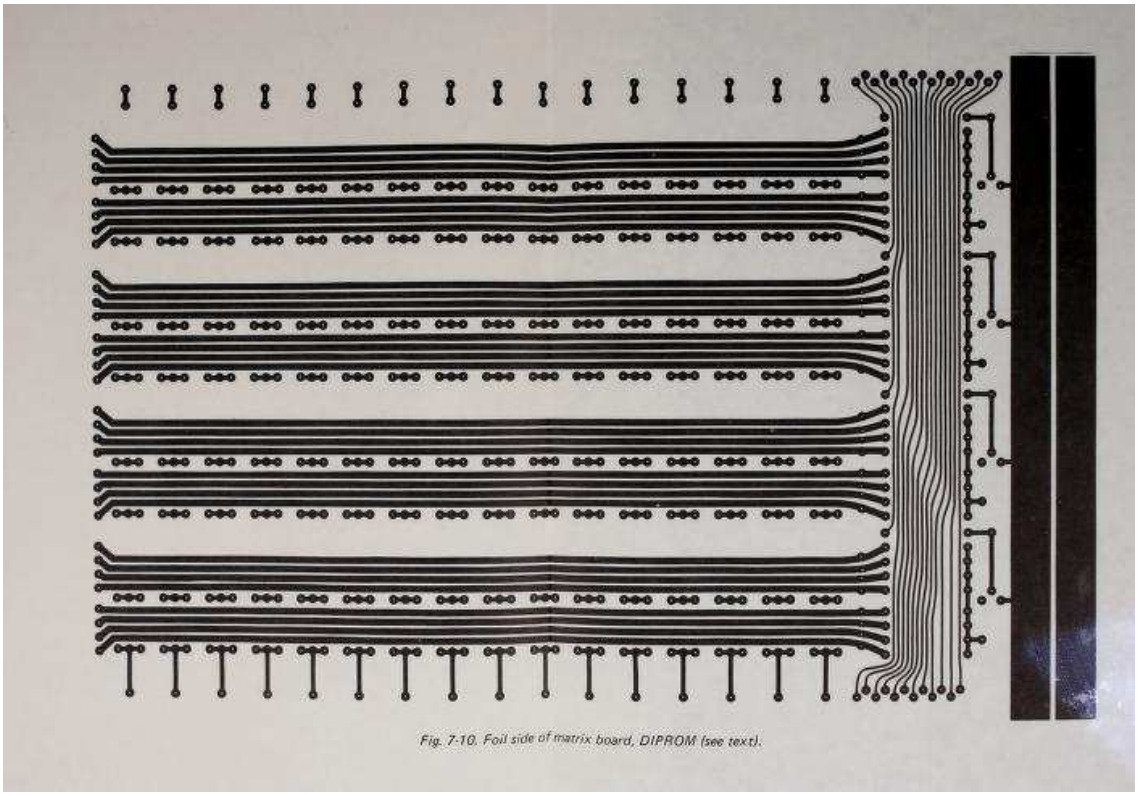
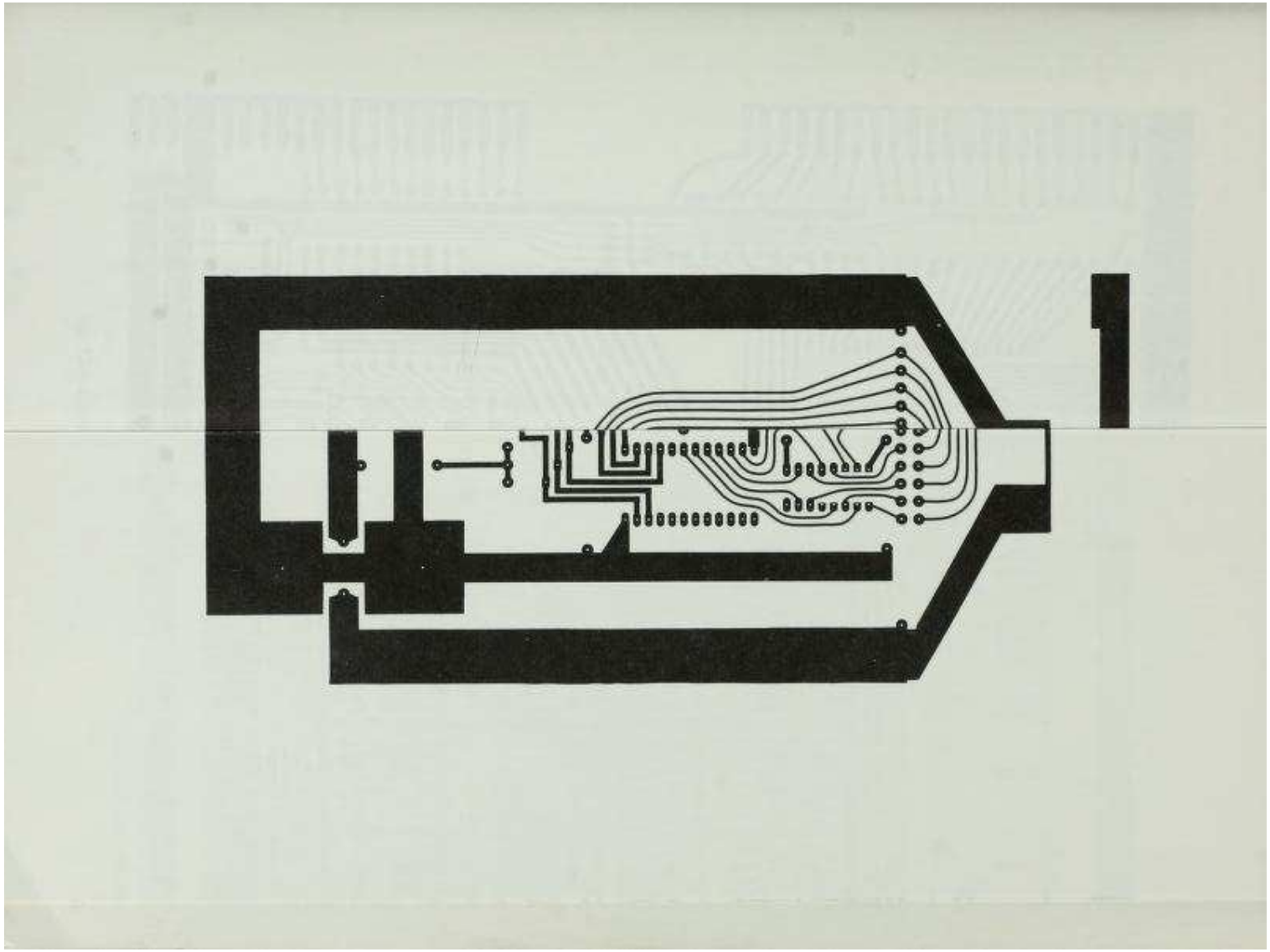


Fig. 7-10. Foil side of matrix board, DIPROM (see text).



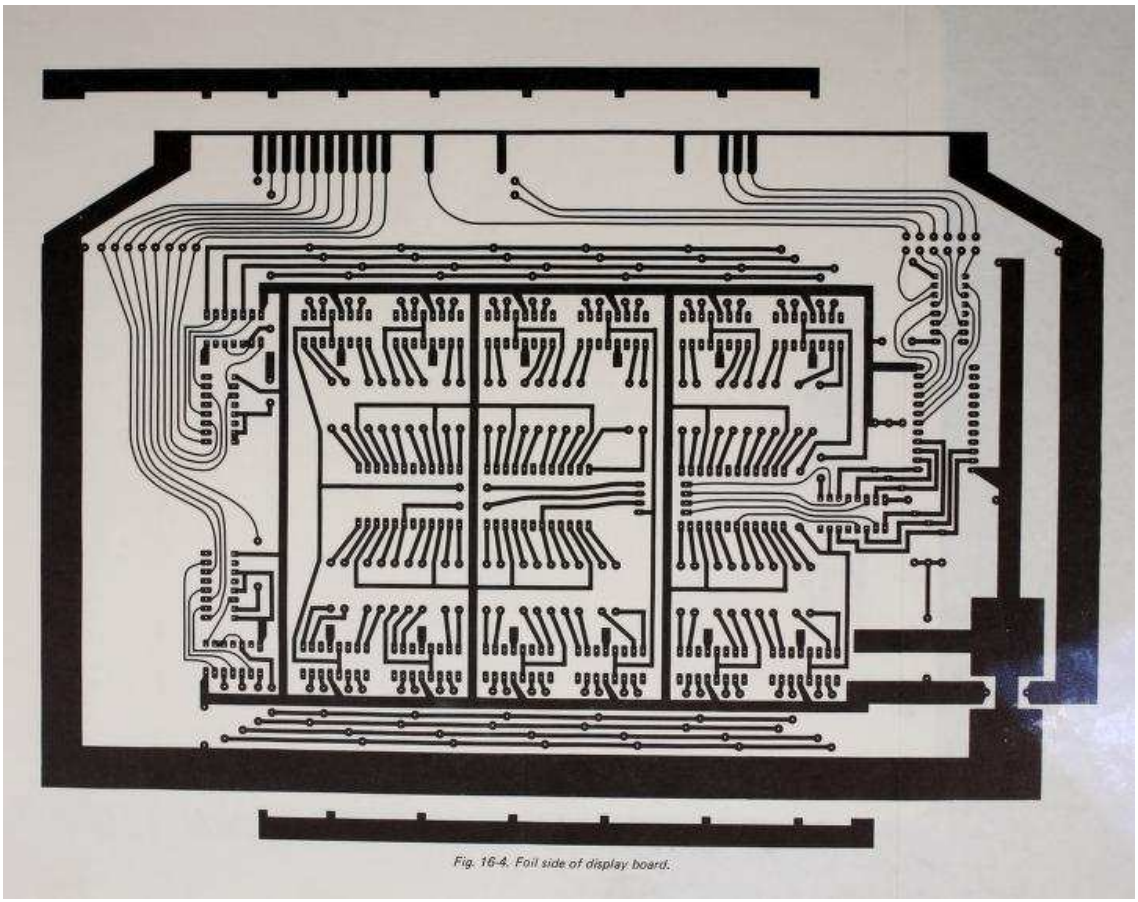


Fig. 16-4. Foil side of display board.

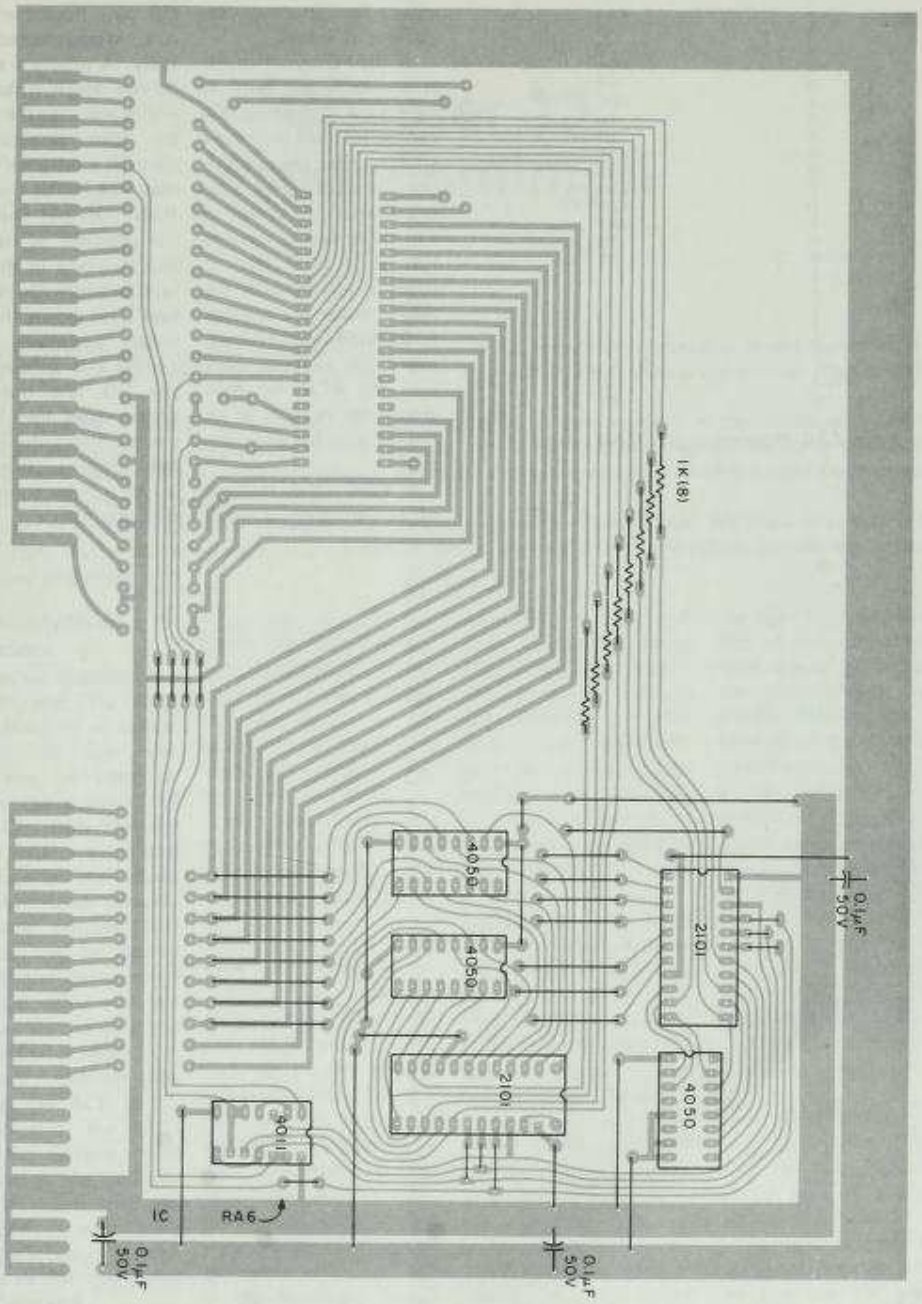


Fig. 12-5. Component layout of the RAM.

LOCATION	INSTRUCTION/CONTENTS
FFE0	LDA
1	42
2	STA
3	00
4	00
5	LDA
6	FF
7	LDA
8	00
9	00
A	STA
B	00
C	82
D	NOP
E	JMP
F	ED
FFF0	FF
I	
I	
I	
I	
I	
I	
FFFC	E0
D	FF

Fig. 12-6. Program to test RAM circuit.

of socket) to see if the proper outputs are produced for various combinations of inputs.

If the PC layout which is shown in this book is used, the remaining ICs can be inserted into their sockets after the preliminary checks. The program which is shown in Fig. 12-6 should then be run as a final test. It loads a number into the ACR, stores the number in location 0000 (RAM), loads the number back into the ACR and then stores the number in an output port for examination.

If some other method of construction is used, the builder may wish to leave the data out and OD pins from the memory disconnected initially. A program should be run which writes several words into the RAM. By then single-cycling through a program which reads the words from the RAM (OD pin tied low) the presence of the correct numbers in the correct locations can be verified. This means that the write operations were carried out properly. If the OD line from the system goes low at the correct times during the single-cycling, then the wiring can be completed with confidence.

Chapter 13
Programming II

In the previous two chapters, we created some loose ends. We added instructions to our repertoire and built a RAM for our system. However, we've not yet explored in depth the capability that these additions provide.

By the time we've finished this chapter, we'll have tied the loose ends together. In the process we'll examine even more instructions, learn some techniques of programming, and see what capabilities the RAM provides. When we've finished, we'll have learned enough about the internal workings of the 6502 and how to program it to take us through this part of the book.

Finally, in this chapter we'll program the system to perform a useful task. We'll use it to key a CW transmitter. Along the way, we'll write programs which test the RAM and which provide the software equivalent of a timer.

Read/Modify/Write Instructions

When we discussed the SHIFT and ROTATE instructions in an earlier chapter, we saw how they can be used to modify the contents of the ACR. Now that we have a RAM, we can take advantage of the added capability which these instructions provide. Not only can they be used to modify the contents of the ACR, they also can be used to modify the contents of a location in memory without disturbing the ACR at all. To do this, the 6502 reads the number which is stored in the location, modifies it and writes the modified number back into the location – hence the name for this group of instructions.

Two other instructions are included in the group. They are the INCREMENT MEMORY (INC) and DECREMENT MEMORY (DEC) instructions. By using them, we can add 1 to or subtract 1 from the contents of a location in the RAM.

We'll use instructions from this group in the programs which we'll write later in this chapter.

Zero-Page Addressing

When we added a RAM to our system we did more than simply make additional instructions available. We now can use an additional method of addressing, as well. This method differs from the others which we've discussed in that only certain locations are accessible when it's used.

The "memory space" of the 6502 contains 65,536 locations. It's convenient to speak of this space as being divided into 256 "pages," each of which contains 256 locations. In that way the eight less-significant bits of an address specify a particular location within a page, while the eight more-significant bits specify the page number.

Each of the eight more-significant bits of the address of each location in the RAM is 0. That is, we've located the RAM in page zero. Because of this we can now use *zero-page* addressing.

Zero-page addressing allows us to specify a limited number of absolute addresses by means of only one eight-bit number, rather than two. When the 6502 encounters an op code which specifies zero-page addressing, it assumes that the eight more-significant bits of the absolute address are all 0. It takes the word which immediately follows the op code to be the eight less-significant bits of the address. Since the entire instruction occupies two words rather than three, less space in memory is required and the instruction is executed more quickly.

Many of the instructions which we've discussed can use zero-page addressing, as shown in Appendix I.

The Index Register

Our next order of business is to discuss an additional pair of registers within the 6502 – the index registers. Each is an eight bit register which we can use for several purposes. We label them the X register (XR) and

the Y register (YR), for easy reference.

The registers are convenient locations in which to temporarily store numbers. For that purpose, there are instructions which load numbers into the XR and YR (LDX, LDY), and which store the contents of the XR and YR (STX, STY). Both *immediate* and *absolute* (including zero-page) addressing may be used with LDX and LDY. Immediate addressing may not be used with STX and STY.

Another way in which information can be exchanged with the XR and YR is via the ACR. Four instructions are available for this purpose, two for each register. For example, the contents of the ACR are transferred to the XR by the instruction TAX, while the contents of the XR are transferred to the ACR by the instruction TXA. Analogous instructions, TAY and TYA, are available for the YR.

Instructions are also available which increment and decrement the number which is contained in either register. For example, if the XR contains the number 89 and the INCREMENT X instruction (INX) is executed, the XR will then contain the number 8A. If the DECREMENT X instruction (DEX) is then executed, the XR will once again contain the number 89. The analogous instructions for the YR are INY and DEY.

For the present, we'll include only two additional instructions that apply to the XR and YR. These are the COMPARE X (CPX) and COMPARE Y (CPY) instructions. Execution of either of these sets or resets bits in the SR in the same way that executing the CMP instruction does (see Chapter 11). Of course, either the XR or YR, rather than the ACR, is involved.

The op codes for these instructions, their abbreviations, and lists of flags which are affected are contained in Appendix I. Their use is illustrated in the programs which we'll write later in this chapter.

Indexed Addressing

The XR and YR are useful as temporary storage locations. However, they serve a much more important purpose. They provide an additional method of addressing. Consider what's involved if we want to store, say, five numbers successively in an output port. A program which will do this is shown in Fig. 13-1.

The program successively stores the numbers which are located at F000, F001, F002, F003, and F004 into the output port by means of a sequence of LDA and STA instructions.

```
LDA  F000
STA  8200
LDA  F001
STA  8200
LDA  F002
STA  8200
LDA  F003
STA  8200
LDA  F004
STA  8200
```

Fig. 13-1. Program to load and store five numbers.

If only five numbers are involved, such a sequence is useable, but what do we do if many numbers are involved? The problem is of more than just academic interest, since we'll encounter it more than once as we write programs. The solution lies in a method of addressing which we'll discuss now.

The instructions which load the numbers into the ACR use absolute addressing. If we had some way to modify the absolute address which is contained in the first LDA instruction, our problem would be solved. We could then use a single STA instruction. After each sequence we could modify the absolute address and return to the start of the sequence. As we might guess, the absolute address can be modified by using either the XR or YR. In addition to immediate and absolute addressing, many of the instructions which we've discussed already can use a form of addressing called *indexed* addressing. It's best explained by means of the program which is shown in Fig. 13-2.

The first instruction, LDX # 0, simply loads 00_{16} into the XR. The key to the program lies in the second line, LDA F000,X. The ,X in this instruction tells us that the address which will be applied to the address bus is formed by adding the contents of the XR to the absolute address which is specified in the instruction. Since the XR is initially set to 00, the first number which is loaded is the number which is stored at location F000.

```
FFC0 LDX # 0
     LDA F000,X
     STA 8200
     INX
     JMP FFC0
```

Fig. 13-2. Improved program to load and store five numbers.

The third line of the program, STA 8200, stores the number in the output port. The fourth line, INX, increments the XR, and the fifth line, JMP FFC0, starts the process again, except that the XR now contains 01. As a consequence, the program loads the number which is stored in location F001 into the ACR, and then stores it in the output port. Unless we intervene, the process will be repeated endlessly. When the XR is incremented to 00 from FF, the number which is located at F000 will again be loaded, however. No carry is generated to the higher-order half of the address.

Assuming that we want to do something other than just load and store the same set of 256 numbers over and over again, we can insert BRANCH and COMPARE instructions in the loop. In that way a branch will occur when the XR contains a predetermined value. Alternatively, we can initially set the XR to the highest

value which will be needed, minus 1. By decrementing the XR instead of incrementing it, we can use just the BEQ instruction. The branch occurs in the sequence when the XR is decremented to 0. This avoids using a COMPARE instruction, saving time and space.

With that, we've discussed all the instructions and methods of addressing which we'll need in this part of the book. Let's now see how we can simplify the task of programming.

Programming in Assembly Language

To this point we've written most of our programs in numerical form. We've used hexadecimal numbers, sparing ourselves of the task of keeping track of a lot of 1s and 0s. However, as our programs become more complex, writing them will be simpler if we use, among other things, the abbreviations which we've been inventing as we've gone along. In the next few paragraphs, we'll systematize our use of abbreviations to the point where we'll not use numbers when we write programs. We'll have to translate the results into numerical form in order to produce something which can be loaded into the system, of course. However, we'll do our thinking in abbreviations rather than numbers.

When we write a program, we may have to specify as many as three things about a typical

```
      LDX # 0
LOOP  LDA NUMBER,X
      STA OPORT
      INX
      JMP LOOP
```

Fig. 13-3. Programming with abbreviations and labels.

instruction. We must specify the op code. We may have to specify the operand or the address of the operand. And in at least a few cases we must specify the address of an instruction itself. We've already invented an abbreviation for each instruction which we've used. Let's further agree that if we want to refer to an address, we'll simply label it with a name.

A simple example which illustrates what's involved is shown in Fig. 13-3. It's the same program which we examined earlier in the chapter.

In the left-most column, we label any location to which we want to refer. In this case, we must refer to the location which contains the LDX instruction, so we'll label it (LOOP).

In the center column, we place the abbreviations for the op codes.

In the right-most column, we place either the operand (immediate addressing) or the labels which stand for the addresses of the operands (absolute addressing). For this program, NUMBER is the address which contains the first number that is to be stored. OPORT stands for 8200, the address of the output port. LOOP is the address in which the LDA instruction starts.

After we've written our program and decided that it likely will do what we require, we can then mechanically translate the program into a list of numbers. To do this, we first count the number of locations which will be required and select a starting address for the program. We number the lines on a sheet of paper and write each line of our program in the appropriate location. We then translate the program into numerical form. If we want to store our example starting at, say, location FFC0, then the result is as shown in Fig. 13-4.

The sort of thing which we've been doing is called "programming in assembly language." Our particular assembly language is made up of the abbreviations and labels which we chose. Writing programs in this way is so much easier than writing in numbers that some sort of assembler program is available for almost any uP on the market.

The input to an assembler program is a list such as that shown in Fig. 13-3. The output from an assembler program is a list such as that shown in Fig. 13-4.

Throughout the remainder of the book, we'll write our programs in assembly language and then translate them into numerical form.

Flowcharting

Programming in assembly language will be a convenience, but it's not the only technique which we can use in streamlining our programming. At this point, we'll discuss a technique called flowcharting and see how it can help us.

Flowcharting provides a way for us to put the essence of our program on paper without getting lost in details. When we flowchart, we write an outline of our program. It bears the same relation to a program as a block diagram bears to the schematic diagram of a circuit.

The flowchart of a very simple program is shown in Fig. 13-5.

By convention, we indicate the beginning and end of a program by rectangles with semicircles attached to the ends. We use ordinary rectangles to indicate arithmetic operations,

LOCATION	INSTRUCTION	CONTENTS
FFC0	LDX #0	A2
1		00
2	LOOP LDA NUMBER,X	BD
3		00
4		F0
5	STA OPORT	8D
6		00
7		82
8	INX	E8
9	JMP LOOP	4C
A		C2
B		FF

Fig. 13-4. Translation into machine code.

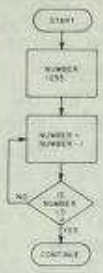


Fig. 13-5. A simple flowchart.

parallelograms to indicate I/O operations and diamonds to indicate that a decision must be made. We indicate the sequence of steps with arrows.

As we can see from the flowchart, the purpose of the program is to store the number 255 in a location and then repeatedly subtract 1 from the number until the result is 0. Such a program will introduce a delay in the execution of a larger program.

A Timer Program

The amount of delay which the program can produce is limited by the time which is required in order to traverse the loop and the number of times the loop is traversed. Depending on the frequency of the clock in the uP, such a program will produce a delay of up to a few milliseconds when run in our system. If we require a longer delay we can cascade the process and insert this program into a second, similar program. The flowchart is shown in Fig. 13-6.

The first step which is shown normally would not be included. However, to check out this particular program, we need some sort of indication that its execution has begun. We don't need to know what that indicator is to write the flowchart, though.

In the next step, a number called TALLY is set to a predetermined value. That value determines how many times the outer loop will be traversed.

A second number, called NUMBER is set to 255 and execution of the inner loop begins. When NUMBER has been decremented to 0, TALLY is decremented by 1. If that reduces TALLY to 0, the indicator is turned on and whatever is signified by "continue" happens. If TALLY is not yet 0, NUMBER is set to 255 and the inner loop is executed again. The entire process is repeated until TALLY is reduced to 0. Several seconds' delay can be introduced by such a program. In effect, we have the software equivalent of a timer.

An assembly language listing of the program is shown in Fig. 13-7. The indicator in this case is an output port. At the start of the program, 00_{16} is stored in the port. After the delay, FF is stored in the port. The program then enters an endless loop.

A Memory-Test Program

A task which remains for us is to test the RAM. We know that the circuit works because we tested it in the previous chapter. However, we've not yet tested all the locations within the 2101s. To do this, we'll write a series of checkerboard patterns into the RAM and see if we can read back what we thought we wrote. The program which we'll use in order to write the checkerboard patterns is shown in Fig. 13-8. No flowchart is shown because the program is quite simple.

The first three instructions store the number AA into location 0000. At this point the ACR also contains the number AA. The next instruction forms the EXCLUSIVE-OR of AA and FF (55), leaving the result in the ACR. The XR is then incremented. Since it doesn't contain 0, the branch backward occurs and the number 55 is stored into location

Fig. 13-6. Flowchart of timer program.

LOCATION		INSTRUCTION	CONTENTS
FFC0		LDA #0	A9
1			00
2		STA OPORT	8D
3			00
4			82
5		LDA COUNT	AD
6			FF
7			FF
8		STA TALLY	85
9			02
A	START	LDA #0	A9
B			00
C		STA NUMBER	85
D			01
E		LOOP DEC NUMBER	C6
F			01
D0		BNE LOOP	D0
1			FC (-4)
2		DEC TALLY	C6
3			02
4		BNE START	D0
5			F4 (-12)
6		LDA #FF	A9
7			FF
8		STA OPORT	8D
9			00
A			82
B		WAIT NOP	EA
C		BMI WAIT	30
D			FD (-3)

NUMBER IS LOCATION 0001
TALLY IS LOCATION 0002
OPORT IS LOCATION 8200
COUNT IS LOCATION FFFF

Fig. 13-7. Assembled listing of timing program.

LOCATION	INSTRUCTION	CONTENTS
FFB0	LDX #0	A2
1		00
2	LDA #AA	A9
3		AA
4	LOOP STA 0000,X	95
5		00
6	EOR #FF	49
7		FF
8	INX	E8
9	BNE LOOP	D0
A		F9
B	WAIT NOP	EA
C	BEQ WAIT	F0
D		FD

Fig. 13-8. Program to write a checkerboard into the RAM.

0001. The EXCLUSIVE-OR of 55 and FF is then formed, leaving AA in the ACR. This is stored in location 0003. The process is repeated until all 256 locations in the RAM are filled. Each even-numbered location then contains AA and each odd-numbered location contains 55. Since $AA_{16} = 10101010_2$ and $55_{16} = 01010101_2$ the RAM contains a checkerboard pattern.

Before we consider how to examine the pattern, let's first note that three other patterns should be used as well. The first of these is obtained by replacing AA with 55 in the second instruction. This merely interchanges the locations of the 1s and 0s in the pattern.

The remaining two patterns are obtained by inserting the instruction STA 01,X after the first STA instruction. A second INX is inserted after the first. We obtain two different "coarser" patterns by using first CC and then 33 as the operand in the LDA instruction.

After any pattern has been written into the RAM, we check its integrity by using the program which is shown in Fig. 13-9. The program is not at all unlike one which we considered earlier. It endlessly stores in the output port the numbers which we wrote into RAM, preserving the order in which they were written.

The CLC and BCC instructions which form the "bottom of the loop" could just as well be replaced with a JMP instruction. However, by using them we can store the program anywhere we please in the DIPROM without rewriting the machine code, since the BCC instruction employs *relative* addressing while the JMP instruction employs *absolute* addressing.

LOCATION	INSTRUCTION	CONTENTS
FFD0	LDX #0	A2
1		00
2	LOOP LDA 0000,X	B5
3		00
4	STA OPORT	8D
5		00
6		82
7	INX	E8
8	CLC	18
9	BCC LOOP	90
A		F7

Fig. 13-9. Program to read a checkerboard from the RAM.

In effect, the program provided us with a digital-signal generator. If the integrity of a checkerboard pattern has been preserved, then a square wave is produced at each terminal of the output port. When examined with an oscilloscope, each square wave should be completely free of jitter. In that regard, it's useful to write FF and 00 into random locations to see the effect which a flaw in the pattern produces. With my scope a flaw produces a faint continuous line across the top or bottom of the square wave.

If an oscilloscope isn't available, we might incorporate the timer function into the program and monitor each terminal (one at a time) by means of an LED indicator. By observing the "blink rate" of the LED, we might be able to spot a flaw. I've not tried that, however.

A CW Message Generator

In what follows, it's assumed that the reader is familar with Morse code. If this isn't so, a basic text on the subject should be consulted.

When we discussed the memory-test program, we described the system as a digital signal generator. We'll carry that thought a little farther now and transform the system into a CW message generator. To begin, let's examine the timing which is involved in sending Morse code.

We can define the durations of dashes and spaces in terms of the duration of a dot. If a dot is taken as one unit of time, than a dash is three units long. The spaces between the dots and dashes within a given letter or character are each one unit long. The spaces between letters or characters are each three units long, and the spaces between words are each seven units long. For example, the timing which is involved in sending "CQ" is shown in Fig. 13-10. Since a CW signal is either on or off, the problem of implementing a CW message generator is essentially one of turning a bit on and off in a predetermined sequence. If that bit is accessible to the outside world, it can be used to drive a keying relay or transistor. The obvious way to proceed, then, is to define the characters C and Q in terms of 1s and 0s. We represent a dot by a single 1 and a dash by three consecutive 1s. We represent a space by the

Fig. 13-10. Timing of CQ.

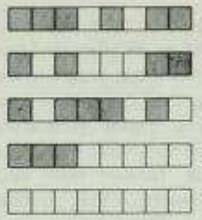


Fig. 13-11. Digital image of CQ.

appropriate number of consecutive 0s. The result is shown in Fig. 13-11.

The pattern of 1s and 0s is broken into four-bit segments, producing the hexadecimal equivalents which are shown. If we store the sequence in memory, we have a digital representation of CQ. If we then designate one bit of an output port as the "active" bit — the bit which will carry information to the outside world — then our task is clear. We must pass the digital representation across the active bit at a relatively slow rate. A flowchart which summarizes what's involved is shown in Fig. 13-12.

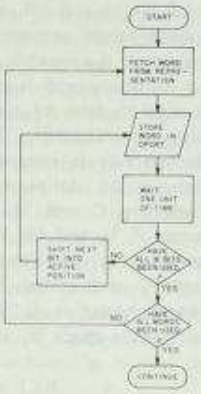


Fig. 13-12. Flowchart of CW message generator.

A word from the digital representation is fetched and stored in the output port. After a unit of time which is equal to the duration of a dot has elapsed, the number of bits which has been used is checked. If not all eight bits have been used, the next bit is shifted into the active position, the result is stored in the output port and a unit of time elapses. The process is repeated until all eight bits have been used. When that happens, the number of words which has been used is checked. If not all have been used, the next word is fetched and the entire process is repeated. This continues until all words have been used.

The delay can be introduced by using the timer program which was discussed earlier. However, in our example we'll use the timer on the I/O board. Further, we'll use bit 7 of the output port at location 8200 to transfer data out of the system. We'll use the control pulse of the port to trigger the timer and we'll connect the output of the timer to bit 7 of the input port at location 9000. In that way, storing a number into the output port will provide information to the outside world and will also start the timer. If we make the period of the timer equal to the desired duration of one dot, then all we have to do is monitor the output of the timer and update the contents of the output port when the timer reverts to its untriggered state Each time the contents are updated the timer is restarted. The program which we'll use is shown in Fig. 13-13. It sends "CQ" once, in response to a reset command.

The XR is set to 0 and the first word of the digital representation is fetched (indexed addressing). The word is stored in a location in the RAM. Another location in the RAM is set to 8 for use as a bit counter.

The word from the digital representation is then stored in the output port. Since the first word is EB, the active bit is turned on. The contents of the input port at location 9000 are repeatedly checked until the most significant bit is 0, indicating that the timer has reverted to its untriggered state. The word from the digital representation is shifted left and the bit counter is decremented. If this doesn't reduce the count to 0, the shifted word is stored in the output port and the process is repeated. If the bit count has been reduced to 0, then the XR is incremented and the result is compared with 5, the number of words in the representation. If not all the words have been used, the next one is fetched and the entire process is repeated. After all the words have been

LOCATION		INSTRUCTION	CONTENTS
FFC0		LDX #0	A2
1			00
2	START	LDA CODE,X	BD
3			B0
4			FF
5		STA WORD	85
6			07
7		LDA #8	A9
8			08
9		STA COUNT	85
A			18
B	FETCH	LDA WORD	A5
C			07
D		STA OPORT	8D
E			00
F			82
D0	WAIT	LDA IPORT	AD
1			00
2			90
3		BMI WAIT	30
4			FB
5		ASL WORD	06
6			07
7		DEC COUNT	C6
8			18
9		BNE FETCH	D0
A			F0
B		INX	E8
C		CPX #5	E0
D			05
E		BNE START	D0
F			E2
E0	FIN	NOP	EA
1		BEQ FIN	F0
2			FD
B0	CODE		EB
1			A3
2			BA
3			E0
4			00

Fig. 13-13. CW message generator program.

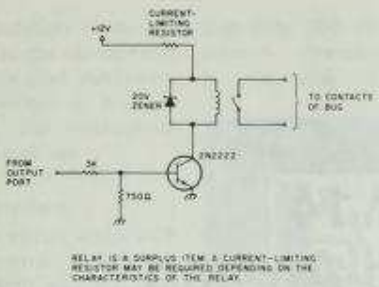


Fig. 13-14. Interface to a QRP transmitter.

used, the program enters an endless loop.

The message can be extended indefinitely depending only on the amount of memory which is available. As each word is added, the counter in location FFDD must be increased by 1.

In my station, the transmitter is a five-watt solid-state rig. The circuit which is used to interface the system to the transmitter is shown in Fig. 13-14.

The 4050 buffer in the output port drives a 2N2222 transistor which, in turn, drives a small relay. The relay contacts are in parallel with the contacts of my "bug."

An advantage of running QRP is that there's not much RF floating around the shack. High levels would probably interfere with the proper operation of the system. Those who contemplate keying a kW rig with the system are advised to purchase a good supply of shielding and bypass capacitors. Some will be needed even with a QRP rig, since the uP clock will cause QRM in the station receiver otherwise.

Chapter 14
The Digital-to-Analog Converter

Until now, we've concerned ourselves almost solely with digital signals. Such signals are always at one or the other of two levels. We've seen that imposing such a restriction can produce very useful results. However, it's a fact of life that we live in an analog world. Voltage levels may be continuously variable, for example. In order to make full use of our system, we'll have to provide a means for it to exchange analog information with the real world.

Since our system is unalterably digital, we won't try to use it to process analog information. Rather, we'll convert the analog information into digital information in order to get it into the system. Conversely, we'll convert digital information to analog information to get it out of the system.

Of the two processes, we'll consider the digital-to-analog (D/A) conversion in this chapter. Along the way, we'll add a converter to our system and implement a programmable signal generator.

Before we go further, it's fair to ask just what it is that a D/A converter does. For our purpose, we describe a D/A converter as a device which accepts a binary number as an input and produces an output which is proportional to the value of that number. Devices are available which combine the necessary functions into a single IC.

The D/A Converter IC

The device which we'll use – a 1408L D/A converter IC – accepts an unsigned eight-bit binary number and produces a current which is proportional to the value of the number. If the number is 0000 0000, the current is 0 mA. If the number is 1111 1111, the current is -2 mA. Between these two extremes, the current is proportional to the magnitude of the number.

The pin configuration of the 1408L is shown in Fig. 14-1.

Pins 2, 3, and 13 provide power connections.

The eight-bit number is applied to pins 5-12, the data pins, while the current is available at pin 4, the output pin.

Discrete components are connected to the remaining pins of the 1408L in order to scale the output and stabilize the converter. We'll not discuss their functions further, but merely remark that we'll follow the manufacturer's rec-

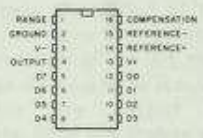


Fig. 14-1. Pin configuration of the 1408-L.

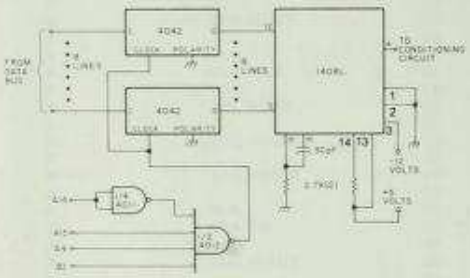


Fig. 14-2(a). Interface to D/A converter.

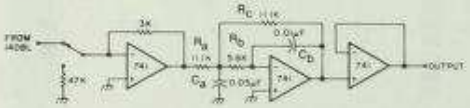


Fig. 14-2(b). Interface to D/A converter.

ommendation with respect to the components that are used. Additional information is available from the appropriate data handbook.

The Interface

The circuit which we'll use in order to interface the 1408L to our system is shown in Fig. 14-2.

It consists of a latch which applies the eight-bit number to the IC and a conditioning circuit which converts the output of the IC into a more useful form.

The eight-bit latch is nothing more than an output port of exactly the same sort as is on the I/O board. Its address is 8010, since address line A4 is dedicated to it.

The output of the IC is applied to a 741 op-amp which is configured as a current-to-voltage converter (CVC). Since the current which is produced by the 1408L varies between 0 and -2 mA, the output of the CVC varies between zero volts and some more positive value. Use of a 3k resistor in the feedback path, as shown, means that the more positive value will be 6 volts.

The switch and resistor at the input of the CVC provide a means to disable the CVC. This will be useful later. If we examine the output of the CVC with an oscilloscope as a series of numbers is stored in the eight-bit latch, we'll find that its output doesn't change smoothly. This is so because the 1408L can provide only 256 different currents (corresponding to the 256 numbers which we can apply). Thus, one of only 256 different discrete voltages can appear at the output of the CVC, and its output can change only in

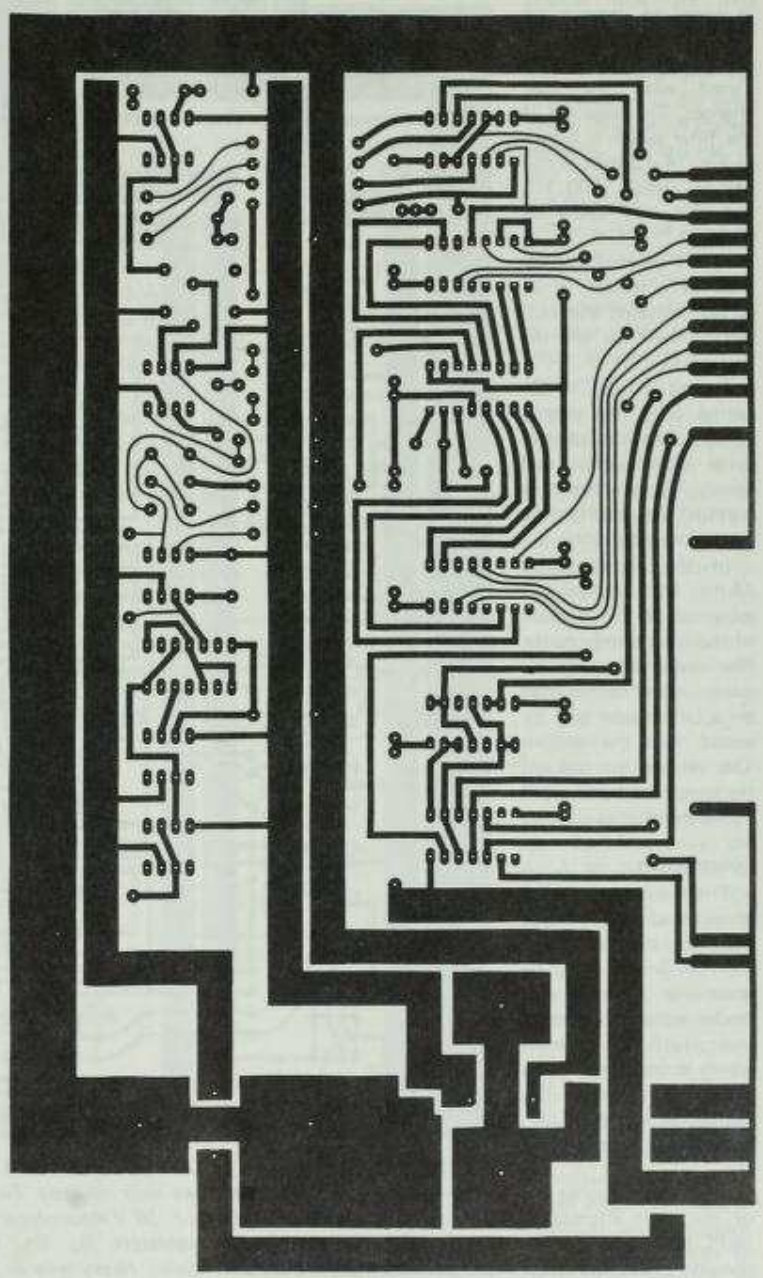


Fig. 14-3. Foil side of D/A converter board.

steps.

Because of this, we apply the output of the CVC to a low pass filter in order to remove the high frequency components which are associated with the step changes. The response of the filter which is shown in Fig. 14-2 begins to roll off at about 1000 Hz, but this may easily be changed.[1]

A 741 op-amp, configured as a buffer, is inserted between the output of the filter and the real world.

5 volt power is obtained from the supply which is on the control panel board. Plus and minus 12 volt power is supplied by appropriate 3-terminal regulators.

In the prototype, a 14-pin DIP socket was mounted at the location of the filter components. The components themselves were then mounted on a DIP header and inserted into the socket. This allows the roll off frequency to be changed by means of plug-ins.

Construction

The foil side of the PC board and the component layout are shown in Figs. 14-3 and 14-4, respectively. The observant reader will see that more than just the circuit which is described in this chapter is included. The remaining portion is discussed in the next chapter.

Those who choose one or the other alternatives to PC board construction shouldn't have any major problems, but none of

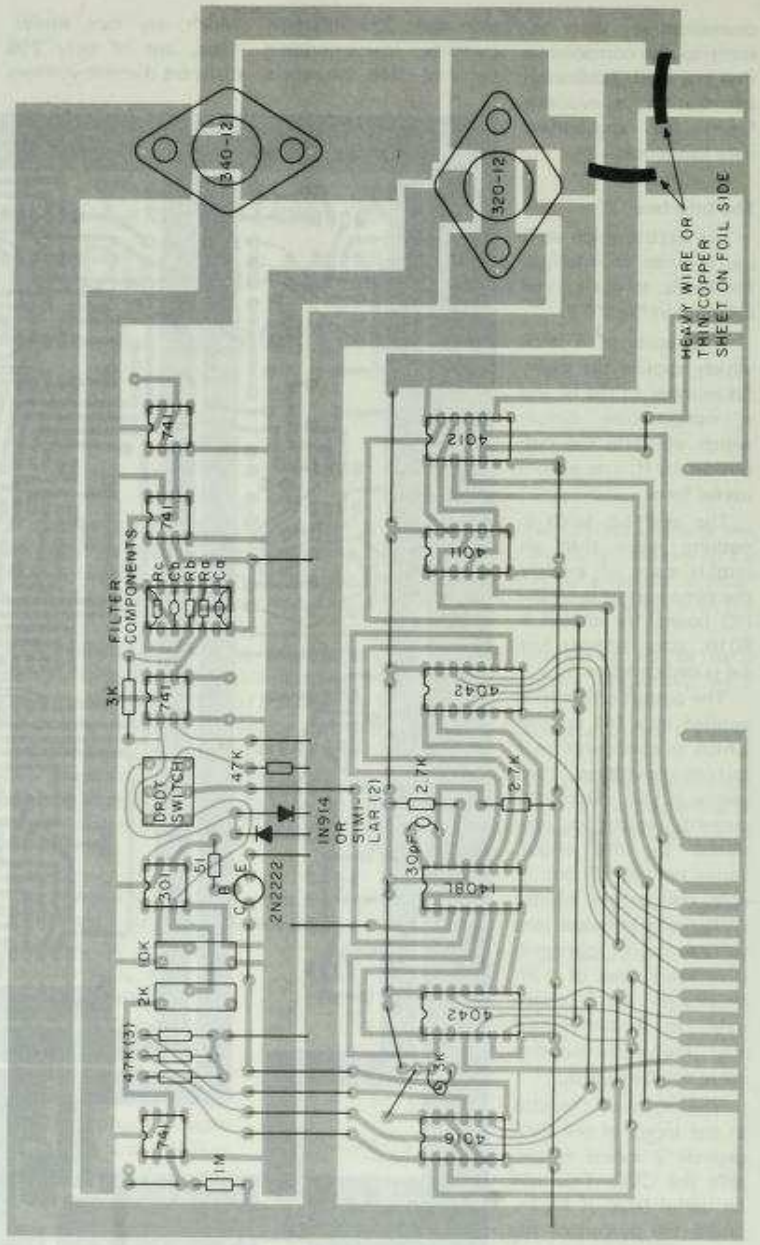


Fig. 14-4. Component side of D/A converter board. Additional components at upper right are A/D converter (see next chapter). Tie 0.1 uF, 50 V disk ceramic; 6.8 uF, 25 V tantalum; and 47 uF, 50 V electrolytic capacitors across outputs of regulators. Use heatsinks on regulators. R_a, R_b, R_c, C_a, and C_b are filter components mounted on DIP header. Heavy wire or thin copper sheet is used to make jumpers for power connections.

the functions were breadboarded during the design of the prototype.

Verifying Proper Operation

Verifying proper operation of the D/A converter is relatively straightforward. The first order of business is to see that the voltage regulators are working properly and that +5 volt power is present.

Once the proper voltages are available we check to see that the uP can write a number into the port which applies the number to the D/A converter IC. This is most easily done by running a simple load-store program and using a logic probe to check the outputs of the port.

We then check out the CVC by temporarily connecting a 10k (or so) resistor to pin 4 of the 1408L socket and monitoring the output of the CVC with a voltmeter. If we connect the free end of the 10k resistor to -12 volts, the output of the CVC should be about +4 volts.

The filter has unity gain at low frequencies and it inverts the sign of the voltage which is applied to its input. The buffer has unity gain, but does not produce an inversion of sign. Thus, when the 10k resistor is connected to +12 volts, the output of the buffer should be +4 volts.

If all is in order, we can check out the D/A converter. With all the ICs in their sockets, we run the program which is shown in Fig. 14-5. It's very simple, but quite effective for our purpose.

LOCATION	INSTRUCTION		CONTENTS
FFF0	LOOP	STX DAPORT	8E
1			10
2			80
3		INX	E8
4		JMP LOOP	4C
5			F0
6			FF

Fig. 14-5. Program to test D/A converter.

The number which the XR contains, whatever it may be, is stored in the port of the D/A converter. The XR is then incremented and its contents are stored in the port. The process is repeated endlessly. As this happens, the number which is stored in the port increases to 255 (relatively slowly) and then suddenly drops to 0, again and again. The result is a sawtooth wave at the output of the CVC. An oscilloscope will verify its presence. After the sawtooth passes through the filter its corners will be rounded significantly. The waveform which appears at the output of the filter should also appear at the output of the buffer.

A Programmable Signal Generator

At this point, we have a working D/A converter. By using the program which is shown in Fig. 14-6, we can use our system as a programmable signal generator.

The program continuously and consecutively stores a series of sixteen numbers in the port of the D/A converter. The way in which the XR is manipulated may seem a bit strange. However, it ensures that the time which elapses between the storage of the sixteenth number and the first number is the same as the time which elapses between the storage of any two other consecutive numbers.

LOCATION	INSTRUCTION	CONTENTS
FFC0	LDX #0	A2
1		00
2	LOOP LDA WAVE,X	BD
3		90
4		FF
5	STA DAPORT	8D
6		10
7		80
8	INX	E8
9	TXA	8A
A	AND #F	29
B		0F
C	TAX	AA
D	JMP LOOP	4C
E		C2
F		FF
FF90	WAVE	80
1		B1
2		DA
3		F5
4		FF
5		F5
6		DA
7		B1
8		80
9		4F
A		26
B		0B
C		00
D		0B
E		26
F		4F

Fig. 14-6. Program to implement signal generator.

The series of numbers which is included in the program generates a sixteen-point sine wave. If other series are used, other waveforms can be generated.

The frequency of the waveform which is generated depends on three things. The most obvious is the frequency of the clock generator in the 6502. Second, the frequency depends on the number of (uP) cycles which elapse between the storage of numbers in the port of the D/A converter. Our program requires eighteen and there doesn't seem to be anything we can do to reduce that number. Finally, the frequency depends on the number of points contained in one cycle of the waveform. The example contains sixteen. Taking all this into account, we calculate that the example will produce a 1.5 kHz sine wave if the clock generator is run at 432 kHz.

We can reduce the frequency of the waveform by including the function of the timer (hardware or software) in the program. In this way we can increase the amount of time which elapses be-

tween the storage of numbers.

Finally, we should note that there is effectively a negative bias on the waveform. This can be eliminated by connecting a resistor between the +12 volt power lead and the summing junction of the CVC. 20k is a good starting value.

Reference

[1] Graeme, J. G., Tobey, G. E., Huelsman, L. P., eds. *Operational Amplifiers: Design and Applications*, Burr-Brown Research Corp., Tucson, Arizona, 1971.

Chapter 15
The Analog-to-Digital Converter

In the previous chapter we considered the problem involved in getting analog information out of a digital system. Our solution lay in constructing a circuit into which the uP can write numbers and which in turn produces a voltage that is proportional to the number.

In this chapter our objective is to solve the opposite problem – how to put analog information into a digital system. In the process, we'll implement the basic function of a digital voltmeter.

An A/D Converter

We might solve the problem in a number of ways. For example, ICs are available which perform the conversion. They accept an analog signal as an input and produce a digital output. However, we can add the function of an A/D converter to our system just by adding a few components to the D/A converter. The basic principle is illustrated in Fig. 15-1.

The analog voltage is applied to the D/A converter via resistor R. This resistor converts the analog voltage to an equivalent current which the D/A converter may be able to sink.

When the analog current is greater than that which the D/A converter can sink, the inverting input of the comparator is pulled positive. When the analog current is smaller than that which the D/A converter can sink, the inverting input of the comparator is pulled negative. The magnitude of the current which the converter can sink depends directly on the number which is applied to the converter.

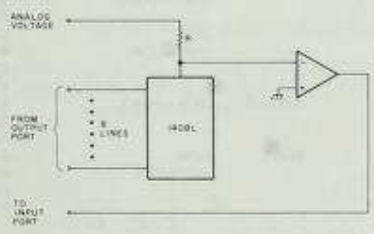


Fig. 15-1. Basis for A/D converter.

Since the non-inverting input of the comparator is tied to ground, the state of the output of the comparator indicates whether the equivalent analog current is larger or smaller than that which the D/A converter can sink.

In an A/D conversion, the uP applies 0000 0000 to the D/A converter and checks the state of the output of the comparator. If the D/A converter is not able to sink the analog current, the uP increments the number which is applied to the converter. This process is repeated until the D/A converter can sink the analog current and the output of the comparator changes state. The number which causes that is the digital equivalent of the analog voltage.

The Circuit

The circuit which we'll use is shown in Fig. 15-2. It's based on a design which originally appeared in a Motorola Application Note.[1]

The analog voltage is applied to the input of a 741 op-amp which is configured as a buffer. The input is also tied to ground via a 1M resistor, which prevents the input from floating. The output of the buffer is applied to the inverting input of the 301 comparator via a 2k variable resistor. The output of the D/A converter is tied to the same point via one half of switch S1.

The non-inverting input of the comparator is held at approximately 0

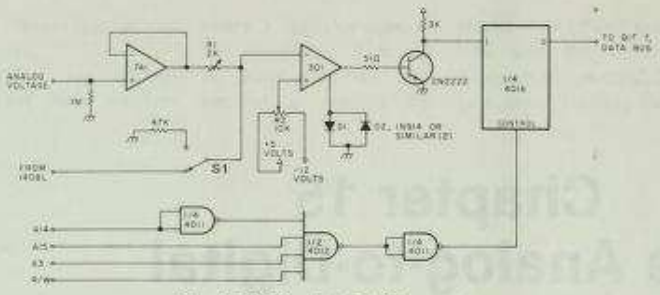


Fig. 15-2. Circuit of A/D converter.

volts, by a 10k variable resistor.

The output of the comparator is clamped within ±0.7 volts or so of ground by diodes D1 and D2. If the inverting input of the comparator is more positive than the non-inverting input, the output of the comparator is negative, and vice-versa.

The comparator drives a 2N2222 transistor which is connected to the data bus via one-fourth of a 4016 quad solid state switch. The 2N2222 transistor may seem superfluous, but it protects the solid state switch, since the output of the comparator can swing far enough negative to damage a 4016. In contrast, the transistor cannot pull the input of the switch negative.

The remaining three switches within the 4016 are available, but aren't required for the A/D converter. The input of each is tied to ground via a 47k resistor in order to prevent damage by static charge.

Construction

Those who constructed the D/A converter using the PC layout which is included in this book have very little work to do in adding the A/D converter. It occupies the unused circuit on that board. The component layout is shown in Fig. 14-4.

Those who chose one or the other alternatives to PC board construction shouldn't have any major problems, but none of the functions were breadboarded during the design of the prototype.

Verifying Proper Operation

If the PC board which was described in the previous chapter is used and the D/A converter works properly, then verifying the proper operation of the A/D converter is straightforward.

The first step is to set R1 to its maximum value and R2 so that its wiper is at 0 volts.

With the control panel and D/A-A/D converter boards (only) in place, we test the 301 comparator and 2N2222 switching transistor (other ICs out of sockets). This is most easily done by temporarily jumpering the inverting input of the comparator to ground via a 10k resistor. We then adjust R2 until the output of the comparator just changes state. If we enable the solid state switches, then bit 7 of the data bus should change state as R2 is adjusted back and forth across the value which toggles the comparator. (Remove jumper after test).

The 741 buffer works properly if the output tracks the input in the range of +10 volts or so.

At this point we can say that all the pieces work properly. We proceed to calibration by replacing the ICs on the D/A-A/D converter board and reinstalling the uP board.

Calibration

To calibrate the converter, we must "zero" it and set the full-scale voltage.

To accomplish the former, we first write 0000 0000 into the D/A converter. Since the input of the buffer draws essentially no current, it's at ground because of the 1M resistor. At this point, then, "zeros" are applied to both legs of the circuit which drives the inverting input of the comparator.

We adjust resistor R2 slightly, first one way and then the other until the input of the solid state switch is pulled low. We then back off the adjustment just enough to pull the input of the solid state switch high again. The converter is now "zeroed."

LOCATION		INSTRUCTION	CONTENTS
FFC0		LDY #0	A0
1			00
2	LOOP	STY DAPORT	8C
3			10
4			80
5		NOP	EA
6		NOP	EA
7		NOP	EA
8		NOP	EA
9		LDA ADPORT	AD
A			08
B			80
C		BMI AHEAD	30
D			04
E		INY	C8
F		JMP LOOP	4C
D0			C2
1			FF
2	AHEAD	STY OPORT	8C
3			00
4			82
5	WAIT	NOP	EA
6		JMP WAIT	4C
7			D5
8			FF

Fig. 15-3. Program to implement A/D converter.

To set the full-scale voltage, we write 1111 1111 into the D/A converter and apply 2.55 volts to the buffer. This should pull the input of the solid state switch low. We then adjust resistor R1 until the input of the solid state switch is just pulled high.

When these adjustments are complete, the range of the converter is 0-2.55 volts. That particular range may seem a bit odd, but in a moment we'll see that it's very useful. For now, let's get our A/D converter "up and running." We do so by means of the program which is shown in Fig. 15-3.

An A/D Conversion Program

The uP applies 0000 0000 to the D/A converter, waits for the output of the comparator to settle, and checks the status of bit seven of the data bus. If the bit is 0, the uP increments the number, waits, and checks the bit again. The process is repeated until the bit is set to 1. The number which causes that is stored in an output port for examination. It's the digital equivalent of the analog voltage.

We can see now why choosing 2.55 volts as the full-scale voltage is so useful. If we apply 2.55 volts, the program will produce 1111 1111 as the equivalent binary number. Of course, $1111\ 1111_2 = 255_{10}$. Since 0 volts will produce 0000 0000 and everything in between is proportional, we can use the A/D converter as a digital voltmeter in a very convenient fashion. All we have to do is divide the decimal equivalent of the binary number by 100. The result is the voltage, in volts, which is applied to the buffer. In the next chapter, we'll construct a digital display and write a program, which will accept the equivalent binary number as an input and display the decimal equivalent.

Reference

[1] Aldridge, D., *Analog-to-Digital Conversion Techniques With The M6800 Microprocessor System, AN-757*, Motorola Semiconductor, Phoenix, Arizona, 1975.

Chapter 16
The Digital Display

In the previous chapter we devised a way to get analog information into our digital system. We now have a device which will accept an analog voltage as an input and provide a binary number as an output. Because of the scaling factor which we chose, the binary number is the same as the magnitude of the analog voltage, in tens of millivolts. We have the basic workings of a digital voltmeter. In this chapter, we'll finish the job. We'll build a digital display and write a program which will display the number in decimal form.

The Display Elements

We'll use FND-510 seven segment (plus decimal point) single digit LED display elements. These are common anode devices which display half-inch-high characters. The pin configuration is shown in Fig. 16-1.

If we connect the common anode to +5 volts, then we can light any segment by grounding the appropriate cathode via a current limiting resistor.

To light a segment at an acceptable intensity, we must pass 2.5 mA or so through it. To light a segment to full brillance requires about 6 mA.

The Interface

Interfacing the FND-510s to our system is relatively straightforward. We provide an eight-bit output port for each, and connect individual bits of the port to individual segments of the display elements via current-limiting resistors. If we write a word into a port, the bits which are set to 0 cause the corresponding segment to be lit.

Except for one minor problem, we could build a set of output ports

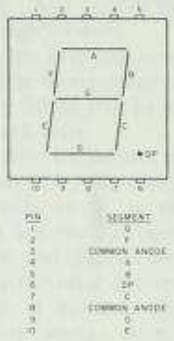


Fig. 16-1. Pin configuration of FND-510.

identical to those which are described in Chapter 8, and proceed directly to the program at the end of this chapter. The problem lies in the amount of current which must be passed through a segment in order to light it properly. We can't reliably sink 2.5-6 mA continuously with a CMOS latch.

The solution is to implement the ports with TTL latches. Low power TTL can sink 2.5 mA and regular TTL can sink 6 mA with ease.

If we simply want to build a display for our digital voltmeter, three such output ports will suffice since the decimal equivalents of the numbers which the A/D converter provides can consist of no more than 3 digits. However, if we write a monitor program for our system, we can make good use of an 6-digit display. With it, we can simultaneously display a 4-digit hexadecimal address and the 2-digit hexadecimal contents of that address.

Our interface, then, will consist of six eight-bit output ports, each implemented with TTL devices. We'll drive individual TTL devices with CMOS buffers to avoid loading the data and address buses. The circuit which we'll use is shown in Fig. 16-2.

Each port consists of a pair of 74L75 quad level-sensitive latches. The outputs of the 74L75s are connected to the cathodes of the FND-510s via 1.5k resistors. Since there is a forward voltage drop of about 1.5 volts across a segment, the current through each is about 2.5

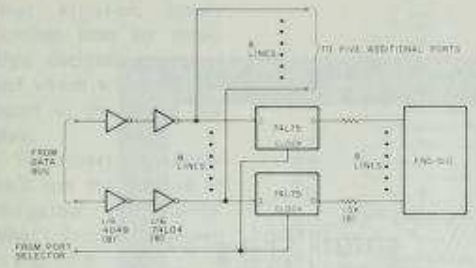


Fig. 16-2(a). Interface to display elements.

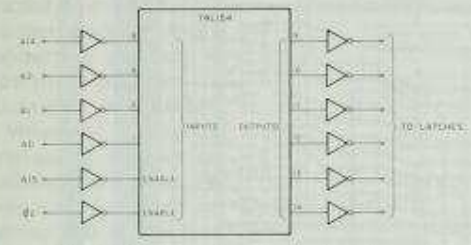


Fig. 16-2(b). Interface to display elements.

mA.

The inputs of each latch connect to on-board data lines which are driven by 74L04 TTL inverter-buffers. The system data bus drives the 74L04s via 4049 CMOS inverter-buffers. In that way, the display presents only a single CMOS load to the data bus. Since two inverters are inserted in each line, there is no net inversion of data.

If the pin designations which are shown for each device are followed, the numbers which must be written into a port in order to display particular characters are as shown in Fig. 16-3.

Control signals for the ports are generated by a 74L154 decoder. Because of this, we need dedicate only three address lines to the six ports. The uP applies four address bits to the 74L154 via a 4049 inverter-buffer. This selects a particular one of the six ports. If A15 is high at the same time, one of the enable inputs of the decoder is taken low. When the Ø2 pulse of the cycle occurs, the remaining enable input is taken low for the duration of the Ø2 pulse, and a write pulse is generated at the appropriate output.

CHARACTER	REPRESENTATION
0	24
1	77
2	1C
3	15
4	47
5	85
6	84
7	37
8	04
9	07
A	06
b	C4
C	AC
d	54
E	8C
F	8E
	FB

Fig. 16-3. Character representations for display elements.

The unique output of the 74L154 is low, but a 74L75 accepts data in response to a high level on its *clock* input. Thus, we insert inverters in the control signal lines. Since CMOS inverters are used, we add pull-up resistors on the output leads of the 74L154.

With the arrangement shown, we've followed our convention that in the address of an I/O port, either A14 or A15, but not both, will be high.

If the pin designations which are shown for each device are followed, the addresses for the ports are 8001 through 8006.

The prototype uses low power TTL devices. Although it has not been attempted, regular TTL devices probably could be substituted throughout. This would permit the use of, say, 750 Ohm pull-down resistors which in turn would light the LEDs brighter. A larger heat sink on the +5 volt regulator likely would be necessary. If the builder doesn't intend to ever try this, then the 74L04 inverter-buffers likely can be eliminated. In that case the 4049 inverter-buffers should be replaced with 4050 buffers. No inversion of data will then be produced.

Construction

The foil side of the PC board and the component layout are shown in Figs. 16-4 and 16-5, respectively *(Editor's note: Fig. 16-4 is a foldout in the center of the book.)*.

Those who choose one or the other of the alternatives to PC board construction do so at their own risk. TTL devices tend to be more sensitive than CMOS devices in such matters as layout and routing of power leads. All breadboarding was done on PC boards.

Verifying Proper Operation

After verifying that the +5 volt regulator is working properly (smoke test), the control circuit and each port should be checked in the same way that those on the I/O board were checked.

A Decimal-Display Program

In the preceding chapter, we implemented the basic function of a digital voltmeter. We now have a device which accepts an analog voltage as an input and provides an eight-bit binary number as an output. Because of the scaling factor which we chose, the binary number is the same as the magnitude of the analog voltage, in tens of millivolts. We'll now write a program which will convert

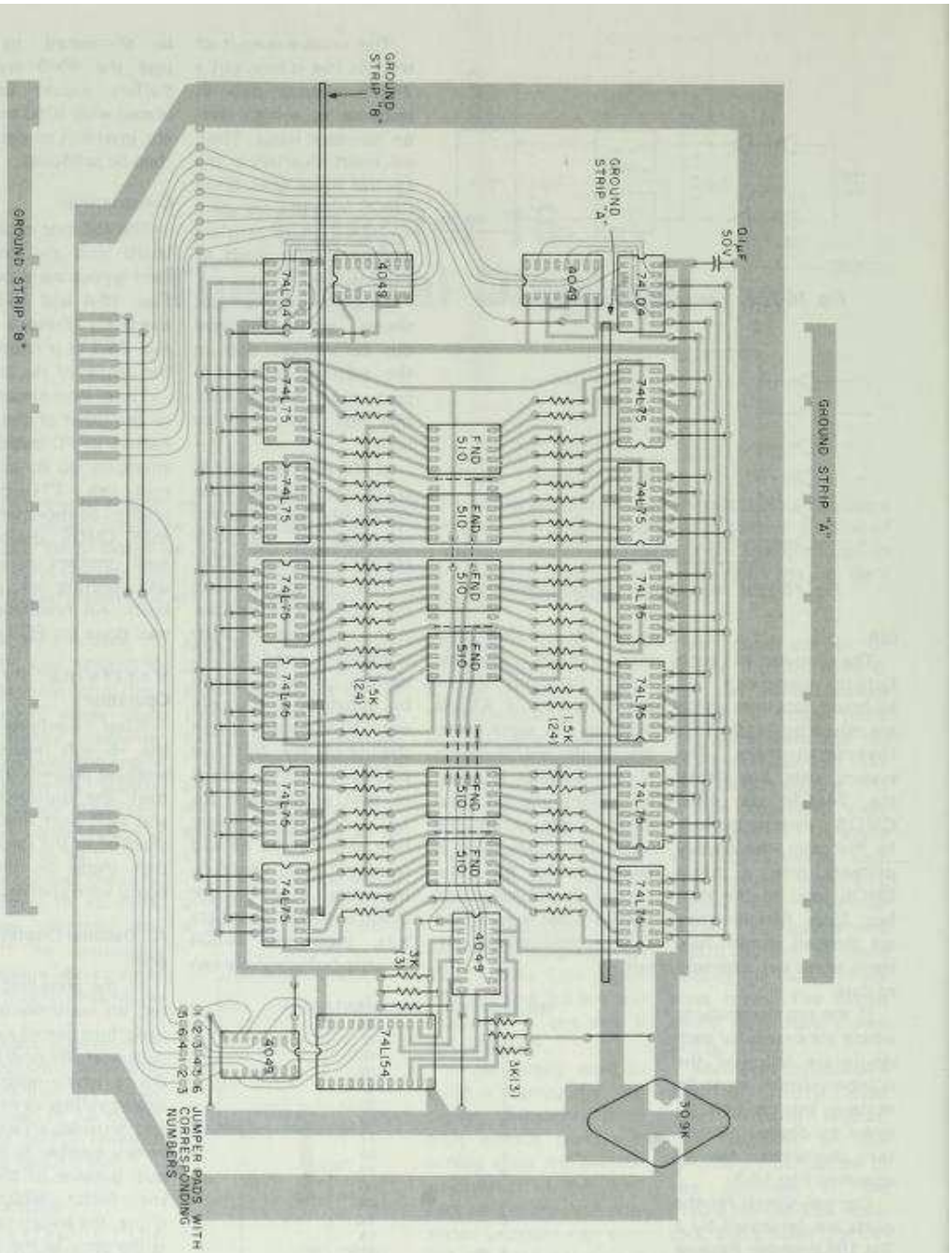


Fig. 16-5. Component side of display board. Tie .1 uF, 50 V disc ceramic; 6.8 uF, 10 V tantalum; and 47 uF, 25 V electrolytic capacitors across output of regulator. Use heatsink on regulator.

that eight-bit binary number into its three-digit decimal equivalent and which will store the result in the digital display.

To convert the number we'll use the method of repeated subtractions. What's involved is shown in Fig. 16-6.

The number 100_{10} (64_{16}) is repeatedly subtracted from the binary number. Each time that a subtraction produces a positive remainder, a counter is incremented.

When the subtraction produces a negative remainder, the process has been carried one step too far, so 100 is added to the remainder. At that point the remainder is positive and less than 100, and the counter shows the number of hundreds which were contained in the original binary number. The equivalent of that count is then stored in a display element.

The entire process is repeated on the remainder, except that the number of tens is counted by repeated subtraction of 10_{10} ($0A_{16}$). The process is carried out yet a third time in order to determine the number of ones which are contained in the second remainder.

The 3 digits which then have been stored in the display form the decimal equivalent of the original binary number.

A program which will do what we've been talking about is shown in Fig. 16-7. The steps which the program carries out in each of the three determinations are similar. A number is fetched, another number is repeatedly subtracted from it and the representation of a count is stored in a display element. Because of this, we can structure the program as a loop which is traversed three times. Of course, different numbers are subtracted and different storage locations are involved during different traverses. However, we accommodate this by counting the number of times that the loop is traversed and using XR-indexed addressing as needed.

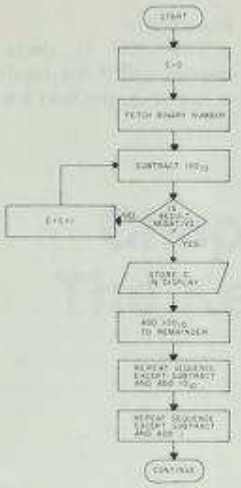


Fig. 16-6. Binary-to-Decimal conversion.

The program starts by fetching the number which is to be converted, and stores it in a location in the RAM. The XR is then set to 3, in anticipation of the three traverses of the loop.

The next two instructions turn off the segments in the display element which is located at 8006. Successive traverses of the loop will do the same for the elements which are located at 8005 and 8004, so that the three unused elements are blanked.

The program then sets the YR to FF, preparing it to count the number of subtractions which will be performed. The number which is to be converted is fetched, and the appropriate subtrahend is repeatedly subtracted until the remainder is negative. In the first series of subtractions the subtrahend is fetched from location FFA9 (FFA6+3). In the subsequent two series of subtractions the subtrahends are fetched from locations FFA8 (FFA6+2) and FFA7 (FFA6+1), respectively.

The value of the subtrahend is added to the negative remainder and the result is saved for the next traverse of the loop.

During the first traverse of the loop, the number of hundreds in the binary number is determined. The representation of that number is fetched (YR-indexing) and stored (XR-indexing) in the display element

LOCATION	INSTRUCTION		CONTENTS
FF80	FIRST	LDA NUMBER	AD
1			FF
2			FF
3		STA TEMP	85
4			70
5		LDX #3	A2
6			03
7		LDA #FF	A9
8			FF
9		STA BLANK,X	9D
A			03
B			80
C		LDY #FF	A0
D			FF
E		LDA TEMP	A5
F			70
90	SUB	SEC	38
1		SBC CONST,X	FD
2			A6
3			FF
4		INY	C8
5		BCS SUB	B0
6			F9
7		ADC CONST,X	7D
8			A6
9			FF
A		STA TEMP	85
B			70
C		LDA DISPLAY,Y	B9
D			E0
E			FF
F		STA ELEMENT,X	9D
A0			00
1			80
2		DEX	CA
3		BNE LOOP	D0
4			E2
5		BEQ FIRST	F0
6			D9
7	CONST		01
8			0A
9			64

NOTE: Program assumes digital representations in Fig. 16-3 are stored in locations FFE0-FFEF.

Fig. 16-7. Program to implement digital voltmeter.

which is located at 8003. In successive traverses of the loop, the representations of the number of tions of the number of ones are stored in the elements which are located at 8002 and 8001, respectively.

The XR is decremented and if the result is not zero, the next traverse of the loop begins. If the result is zero, the program enters an endless loop of instructions.

Chapter 17
The Keyboard

At this point, we have a system which provides a number of useful functions. Of course, it has some obvious limitations. One of these is that we have no way as yet to put information into the system rapidly. We'll remedy that by adding a keyboard and the necessary interface in this chapter. Once the keyboard is working properly, we'll program the system to function as a CW typewriter.

The Keyboard

Most keyboards are fundamentally nothing more than a set of labeled SPST push-button switches, each of which is normally off. Typical keyboards that are available to the hobbyist have sixty-four keys, give or take a few. For our system, we'll implement a keyboard which provides all the letters of the alphabet, the decimal digits, and a few other characters.

The Interface

Our task is to devise a circuit and accompanying program which will accept a relatively large number of switch closures as inputs and provide the appropriate ASCII representations as outputs.

There are at least two ways in which we might go about this. One is to use one of the commercially available keyboard encoder ICs such as the HD0164. In fact, that's really not such a bad method to use. However, we'll use a method which depends on software for the encoding function so that we can improve our programming skills.

The basis of our method is illustrated in Fig. 17-1.

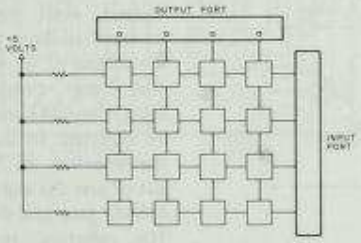


Fig. 17-1. Basis of a keyboard encoder.

Four lines are pulled high by means of pull-up resistors. These lines are connected to an input port. A second set of four lines is connected to an output port into which the number 0000 has been written. A switch is located at each intersection. If one of the switches is closed, one of the terminals of the input port is pulled low. If the number which is applied to the input port is saved, we can identify the row in which the switch is located, since the corresponding bit is 0. We can determine in which column the switch is located by using a similar technique. While the key is pressed, we apply the numbers 1000, 0100, 0010, and 0001 successively via the output port, and save the number which restores the original contents (1111) of the input port. Since we can identify the row and column in which the switch is located, we've unambiguously located the switch.

Our interface is similar, except that an 8 x 8 matrix is involved since our keyboard contains sixty-four keys. Then too, we'll find that it's convenient to dedicate individual bits of a second input port to the SHIFT key and to a line which indicates that a key has been pressed.

In part, the program

which we'll write to accompany the interface will service the I/O ports as described. That portion of the program is relatively straightforward. If we're clever about the way in which we assign locations in the matrix to the various keys, decoding the numbers will be very simple. What's involved is illustrated in Fig. 17-2.

The figure consists of the letters of the alphabet, the decimal digits, and other characters arranged in an 8 x 8 matrix according to three-bit portions of their ASCII representations.

This particular arrangement has a very useful property. If we move a character down a space at a time to the bottom row, we generate the three less significant bits of the ASCII representation of the character if we merely count the number of moves. For example, seven moves are required in order to place the letter W in the bottom row. The binary equivalent of 7 is 111, the three less significant bits of the ASCII representation of W. An analogous process, involving moves to the right, will generate the three next more significant bits of the ASCII representation of W, 010. Combining the two produces 010111, the six-bit ASCII representation of W.

In order to use this arrangement, we'll place the various key switches at the corresponding intersections in the matrix.

Fig. 17-2. ASCII matrix.

We'll connect a terminal from each of the key switches which correspond to characters in column 000 and tie the combination to bit 7 of the output port. We'll connect those in column 001 to bit 6, and so on. Similarly, we'll connect the remaining terminal from each of the key switches which correspond to characters in row 000 and tie the combination to bit 7 of the input port. We'll connect those in row 001 to bit 6, and so on.

In order to decode the identifying numbers from such a matrix all we have to do is shift each number until the unique bit is in the most significant position, an easily detected condition. Counting the shifts and combining the two results produces the ASCII representation.

There are two out-of-place characters in the matrix. The LINE-FEED and CARRIAGE-RETURN keys should be at other intersections, but those intersections are already occupied. The problem is that we really should use seven-bit representations. We'll program our way around the problem, though. The circuit of our interface is shown in Fig. 17-3.

The eight lines which correspond to the columns of the matrix are driven by an output port via 4049 inverter buffers. One terminal of each of several key switches is connected to each of these lines. The other terminals are connected to the eight lines which correspond to the rows of the matrix. These lines are pulled high by 10k resistors and drive an input port of the system via 4049 inverter buffers. These lines are also connected to the inputs of an eight input NAND gate. The NAND gate drives a 4047 timer which, in turn, drives bit 7 of a second input port. Two 4049 inverter buffers also are included in this line. With the components shown, the timer produces a delay of about 50 milliseconds. We'll use the delay in order to "debounce" the key switches.

The SHIFT key drives bit 6 in the second input port. Pressing the key pulls the line low.

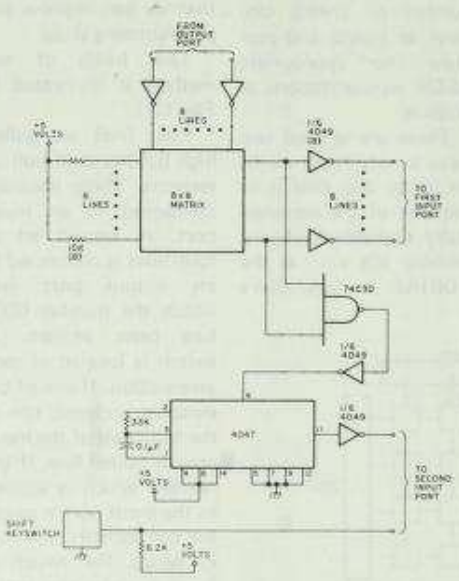


Fig. 17-3. Interface to keyboard.

Construction

The foil side of the PC board and the component layout are shown in Figs. 17-4 and 17-5, respectively.

A PC board likely is unnecessary in this case. Indeed, even the 4049 inverter buffers may be unnecessary, as well. If the timer program is used, the 4047 timer can be omitted. In that case the interface could consist of just the eight-input NAND gate and the eight 10k resistors. However, the prototype was built

with another, unrelated purpose in mind, so the "extra" components were included.

Verifying Proper Operation

Probably the simplest way to verify that the keyboard and interface are working properly is to simulate what the system will do later. First, the lines which would normally be connected to the output port should be connected to +5 volts (via 1k resistors). Then, if the lines which would normally be connected to the input ports are monitored, the effect of pressing various keys can be seen.

Pressing a key should trigger the timer. However, this will likely go unnoticed unless a higher value resistor is temporarily substituted into the timer circuit.

If one of the lines which would normally be connected to the output port is connected to ground, rather than +5 volts, the keys in the corresponding column in the matrix should produce no effect when they are pushed.

A CW Typewriter

At this point, we have a working keyboard. However, to use it we'll have to do some programming. Our final goal is to program the system to work as a CW typewriter. As we go about the task, we'll find that the final program is rather long. Because of this, we'll break it into three separate programs. The first part will generate input for the second and the second will generate input for the third.

The first program which we'll consider services the I/O ports and captures the identifying numbers which correspond to the key which is pressed.

The output port which is located at 8100 provides the signal to the input of the 8 x 8 matrix, while the input port which is located at 9000 accepts the signal from the output of the matrix. The KEY PRESSED line is connected to bit 7 of the input port which is located at A000. The SHIFT key is connected to bit 6 of that port. With those assignments in mind, we can proceed with the program. A flowchart and an assembly language listing are shown in Figs. 17-6 and 17-7, respectively.

The program starts by applying 00_{16} to the matrix. (It actually writes FF into the output port to accommodate the inverters in the lines to the matrix.) The program

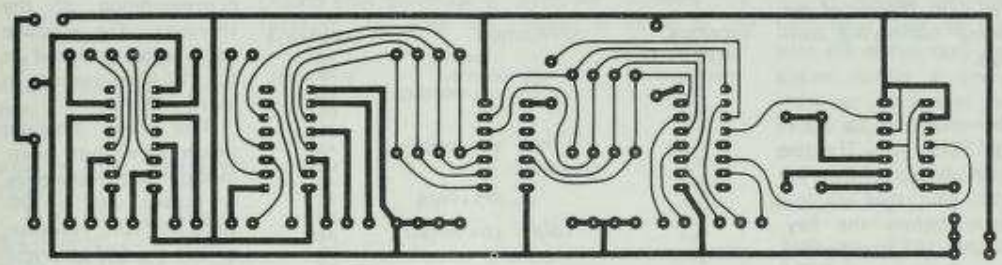


Fig. 17-4. Foil side of keyboard interface board.

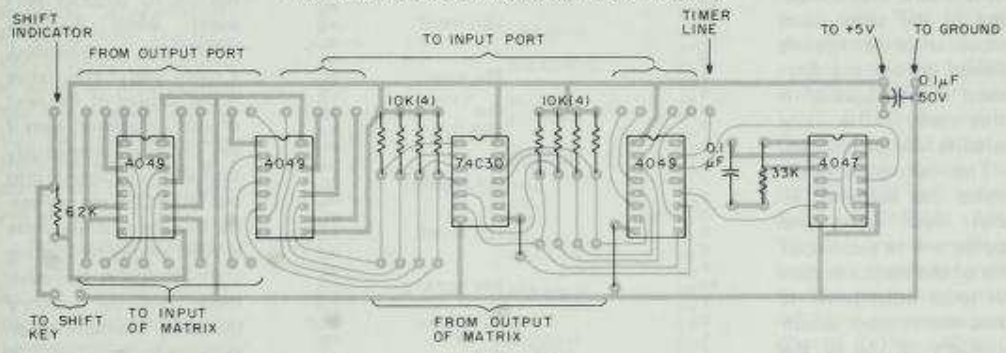


Fig. 17-5. Component side of keyboard interface board.

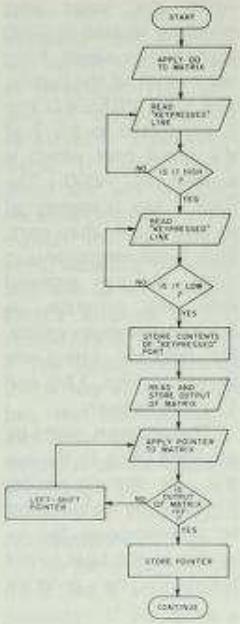


Fig. 17-6. Method of capturing identifying numbers.

then monitors the KEYPRESSED line. The line must be low, then go high, and then go low again before the keyboard is interrogated. Since the line is driven by the timer, this requirement means that the period of the timer must elapse after a key is pressed before the keyboard is interrogated. In that way, the key switches are debounced.

The program then stores the contents of both input ports and applies a 1 to each input line of the matrix in turn (0s to all other lines). It saves the number which produces all 1s at the output of the matrix.

In summary, the program responds to a pressed key and stores the information which indicates which character key was pressed and whether or not the SHIFT key was simultaneously pressed. The machine code which is listed in Fig. 17-7 stores in location 00FC the number which indicates whether or not the SHIFT key was pressed. It then stores the low-order and high-order identifying numbers at locations 00F7 and 00FF, respectively. Examples of the numbers which are stored are shown in Fig. 17-8.

Let's now examine the program which decodes the two identifying numbers into the six-bit ASCII representation of the character. A block diagram and assembly language listing are shown in Figs. 17-9 and 17-10, respectively.

	CONTENTS OF STORAGE LOCATION		
KEY	KEYPRESSED (FC)	LOW (F7)	HIGH (FF)
9	0100 0000	0100 0000	1111 1110
9, SHIFT	0000 0000	0100 0000	1111 1110
W	0100 0000	0000 0001	1101 1111
W, SHIFT	0000 0000	0000 0001	1101 1111

Fig. 17-8. Typical identifying numbers.

The program sets a counter to zero and fetches the low-order identifying number. If bit 7 is 1, indicating that no additional decoding is necessary, the count (0) is saved and the program proceeds to decode the second identifying number. However, if bit 7 is not 1, the identifying number is shifted left repeatedly until it is. The count is incremented each time the number is shifted, and the count is saved.

The process is repeated on the high-order identifying number. The high-order count is shifted left three times and ORed with the low-order count. The result is the six-bit ASCII representation of the character. The machine code which is listed in Fig. 17-10 stores this in location 00FB.

The two programs which we've just examined can be simultaneously loaded into a 128-word DIPROM. However, to add the CW generator program as well requires additional space (assuming an IC encoder isn't used). While the DIPROM can be expanded, a more useful alternative is the subject of the next chapter. However, even if we can't simultaneously load all the programs into our DIPROM, we can verify that the machine code is valid simply by loading a six-bit ASCII representation into the memory by using a simple load/store program. The CW generator program can then use that as input. With this in mind, let's

LOCATION	INSTRUCTION		CONTENTS
FF80		LDA #FF	A9
1			FF
2		STA MOPORT	8D
3			00
4			81
5	LOOP 1	LDA MIPORT	AD
6			00
7			A0
8		BPL LOOP1	10
9			FB
A	LOOP 2	LDA MIPORT	AD
B			00
C			A0
D		BMI LOOP2	30
E			FB
F		STA LOW	85
90			FC
1		LDA IPORT	AD
2			00
3			90
4		STA HIGH	85
5			F7
6		CLC	18
7		LDA #FF	A9
8			FF
9	ROTATE	ROL	2A
A		STA MOPORT	8D
B			00
C			81
D		LDX IPORT	AE
E			00
F			90
A0		BNE ROTATE	D0
1			F7
2		STA HIGH	85
3			FF
4	WAIT	NOP	EA
5		BEQ WAIT	F0
6			FD

Fig. 17-7. Program to capture identifying numbers.

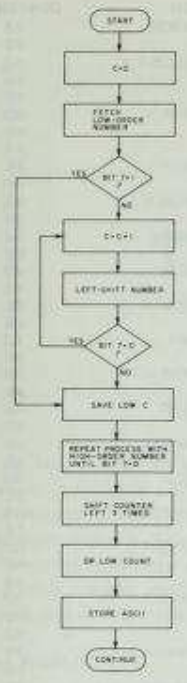


Fig. 17-9. Method of decoding identifying numbers.

examine the generator program.

In writing this generator program, the problem is to convert the six-bit ASCII representation of a character into the corresponding representation in Morse code. The solution is similar to the way in which we implemented the CW message generator. We fetch an image from memory and pass it across one bit of an output port.

The hardware is set up in exactly the same way as for the CW message generator. However, the software must obviously be different. We can't simply store a message in memory and sequentially fetch eight-bit portions of the image since there is no predetermined image. What we can do is store the images of all the characters that we'll want to use and fetch each as it's required. This is most easily done by storing the representations as a table in memory and using the ASCII representation as the offset in indexed addressing of that table.

We have a problem if we try to store images of the sort which we used for the CW message generator, however. The images of many characters each require more than eight bits of space. The solution is to store a more compact image, which the program can expand after the image is fetched. In such an image, we represent a dash by a 1 and a dot by a 0. For example, the eight-bit image of "error" is 0000 0000. In this case, all eight bits are used. Of course the eight-bit images of E, I, S, H, and 5 are also 0000 0000, so that if we use such a representation, we must also provide a counter which indicates how many bits are to be used. For example, if the input to the program is the ASCII representation of H, the program fetches 00_{16} from a table of images and 04 from a table of counters. The sets of images and counters which we use are shown in Fig. 17-11.

The way in which the program expands the images is shown in the flowchart in Fig. 17-12.

The program fetches and stores in temporary locations the representation and the counter from the tables. It then fetches the representation (WORD). If the most-significant bit is 1, it stores "E0" in IMAGE and sets the index to 4. The more-significant four bits of E0 are then sent to the outside world. The effect is to send a dash and a single space. If the most-significant bit of WORD is a 0, the result is similar, except a dot and a single space are sent. A shift-left is performed on WORD so that the next bit can be examined, and the entire process is repeated until the required number of bits have been used. In this way, the dashes, dots and spaces within a single character are sent in the proper sequence with the proper timing.

The program listing is shown in Fig. 17-13.

LOCATION		INSTRUCTION	CONTENTS
FFB0		LDX #0	A2
1			00
2		LDA LOW	A5
3			F7
4		BMI DOWN1	30
5			05
6	SHIFT1	INX	E8
7		ASL LOW	06
8			F7
9		BPL SHIFT1	10
A			FB
B	DOWN1	STX LOW	86
C			F7
D		LDX #0	A2
E			00
F		LDA HIGH	A5
C0			FF
1		BPL DOWN2	10
2			05
3	SHIFT2	INX	E8
4		ASL HIGH	06
5			FF
6		BMI SHIFT2	30
7			FB
8	DOWN2	TXA	8A
9		ASL	0A
A		ASL	0A
B		ASL	0A
C		ORA LOW	05
D			F7
E		STA ASCII	85
F			FB
D0	WAIT	NOP	EA
1		BMI WAIT	10
2			FD

Fig. 17-10. Program to produce six-bit ASCII representations.

The program listing is shown in Fig. 17-13.

Tidying The Six-Bit ASCII

There are a few loose ends remaining to be dealt with. The first of these concerns the encoding of the CARRIAGE RETURN and LINE FEED functions. Our arrangement decodes a carriage return to 011001 (19) and a line feed to 011010 (1A). These representations bear no systematic relationship to the respective ASCII representations, 001101 (0D) and 001010 (0A). We could most easily solve the problem by writing a program

CHARACTER	IMAGE	COUNTER
A	40	02
B	80	04
C	A0	04
D	80	03
E	00	01
F	20	04
G	C0	03
H	00	04
I	00	02
J	70	04
K	A0	03
L	40	04
M	C0	02
N	80	02
O	E0	03
P	60	04
Q	D0	04
R	40	03
S	00	03
T	80	01
U	20	03
V	10	04
W	60	03
X	90	04
Y	80	04
Z	C0	04
1	78	05
2	38	05
3	18	05
4	08	05
5	00	05
6	80	05
7	C0	05
8	E0	05
9	F0	05
0	F8	05
.	54	06
,	CC	06
?	30	06
–	88	05
AR	50	05
SK	14	06
BK	8A	07
ERROR	00	08

Fig. 17-11. Images and counters for CW typewriter.

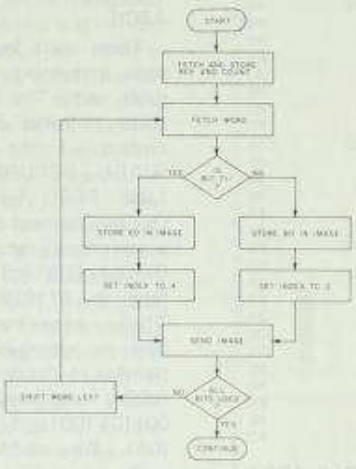


Fig. 17-12. Method to expand and send image.

LOCATION		INSTRUCTION	CONTENTS
FF80		LDX ASCII	A6
1			FB
2		LDY COUNT,X	AC
3			D5
4			FF
5		LDA REP,X	BD
6			B5
7			FF
8		STA WORD	85
9			F0
A	START	LDA WORD	A5
B			F0
C		BPL DOT	10
D			08
E		LDA #E0	A9
F			E0
90		STA IMAGE	85
1			F9
2		LDX #4	A2
3			04
4		BNE AHEAD	D0
5			06
6	DOT	LDA #80	A9
7			80
8		STA IMAGE	85
9			F9
A		LDX #2	A2
B			02
C	AHEAD	LDA IMAGE	A5
D			F9
E		STA OPORT	8D
F			00
A0			82
1	CHECK	LDA IPORT	AD
2			00
3			90
4		BMI CHECK	30
5			FC
6		ASL IMAGE	06
7			F9
8		DEX	CA
9		BNE AHEAD	D0
A			F2
B		ASL WORD	06
C			F0
D		DEY	88
E	WAIT	BNE START	D0
F			D9
B0		BEQ WAIT	F0
1			FC

Note:
Program assumes that CW representations and counts are stored starting at locations FFB5 and FFD5, respectively.

Fig. 17-13. CW output program.

which compares the incoming six-bit ASCII with 19 and 1A and then substitutes 0D or 0A in their places.

We could most easily incorporate the SHIFT function by means of a program that would EXCLUSIVE-OR 10 with the six-bit ASCII representation. This complements bit four of the representation, producing the six-bit representation of the "shifted" character.

Finally, we could generate bit seven of a representation by means of a program which sets it to 1 if the six-bit representation is 1F or below, and to 0 if the six-bit representation is 20 or above.

Chapter 18
The Read-Only Memory

At this point, we've used the system in several applications. Each has involved writing a program and loading it into the DIPROM for use. While that device is an excellent aid to learning and very useful in writing and debugging short programs, once a program works properly, the advantage of the DIPROM almost becomes a disadvantage. If the program is more than a few instructions long, loading it can be a chore. Then too, unless we're careful, we may insert a plug-in incorrectly and find that the program can't be run at all.

In this chapter we'll add a piece of hardware which provides a permanent storage place for our programs. It won't provide new uses for the system, but it will make the system itself easier to use.

The ROM IC

The device which we'll use is a 1702A erasable, programmable, read-only memory (EPROM). A single 1702A will hold 256 eight-bit numbers. The board which we'll build will hold up to four 1702As so that our ROM will hold up to 1024 numbers.

The pin designations of the 1702A are shown in Fig. 18-1.

We can divide the pins according to their functions. For example, +5 volt or -9 volt power is applied to several of the pins, as shown.

An eight-bit address is applied to pins 1-3 and 17-21. If the CHIP SELECT, pin 14, is taken low, the eight-bit number which is stored at that address appears at pins 4-11. If pin 14 is taken high, internal solid state switches are turned off, disconnecting pins 4-11 from the rest of the memory.

Programming The EPROM

While the EPROM and DIPROM have features in common, ease of programming isn't one of them. To program an EPROM requires some fairly expensive equipment. Then too, unless the EPROM has never been programmed (in which case each bit is a 0), its previous contents must be erased. This is done by exposing the IC to high intensity ultraviolet (UV) light. The UV energy passes through the transparent quartz lid of the 1702A and impinges directly on the MOS cells within, erasing their contents. After the EPROM has been erased, it is programmed by a complicated sequence of signals, some of which are 48 volts negative. The output leads serve as input leads during programming.

Fig. 18-1. Pin designations of the 1702A.

Fortunately, we don't have to worry about programming our EPROM. Several mail-order houses provide a programming service (Appendix II). All we have to do is specify what numbers we want stored in what locations.

The Interface

The circuit which we'll use in order to interface the 1702As to our system is shown in Fig. 18-2. The six higher-order address lines (A10-A15) drive a 4050 buffer. Address lines A10-A13 are also applied to a 4049 inverter-buffer. Any combination of A10-A13 and the individual complements can be applied to inputs of the 74C30 NAND gate via jumpers. These jumpers provide the means to specify the locations in memory which the ROM will occupy. A14 and A15 are applied to inputs of the NAND gate in keeping with our convention that both A14 and A15 will be high in any address which involves ROM.

The 74C30 generates the CONTROL signal (more accurately, its

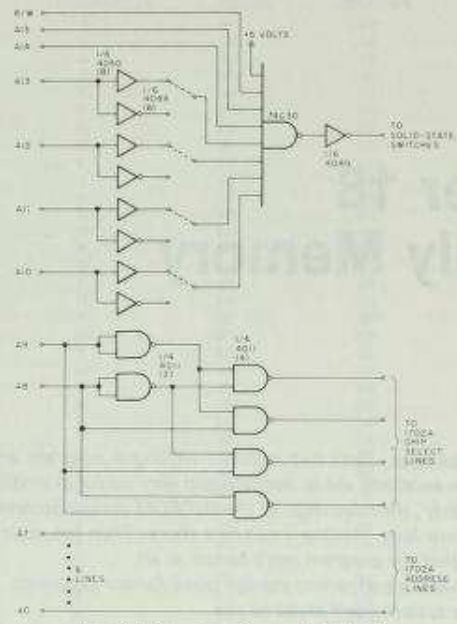


Fig. 18-2(a). Interface to EPROM.

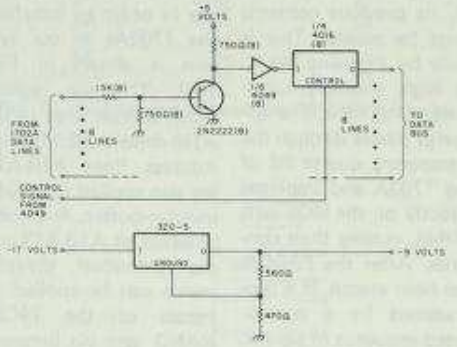


Fig. 18-2(b). Interface to EPROM.

complement) for the on-board solid state switches which route data to the system data bus. A 4049 provides the signal itself.

By means of the jumpers in the address lines, we can interchange the locations of the DIPROM and EPROM at will. In that way we can test a program in the DIPROM while using the exact locations which the program will occupy in the EPROM. However, when we're doing that we may inadvertently locate both ROMs at the same addresses. If that happens, the output buffers or solid state switches of one or the other may be damaged. To avoid that, we jump the complement of the control signal for the DIPROM (control panel) to the unused input of the 74C30 on the EPROM board. In that way both ROMs can never be active at the same time.

The eight lower-order address lines are applied to each of the four 1702As.

Chip select signals are generated by decoding address lines A8 and A9 to produce four unique outputs.

The respective output lines from the 1702As are tied together via on-board DATA lines. These lines in turn drive 2N2222 transistors. The transistors drive 4049 buffers which apply data to the data bus via solid state switches.

The transistors protect the inputs of the buffers from the fairly large negative voltages, -2.5 volts, which can appear at the outputs of the 1702As.

+5 volt power for much of the board comes from an on-board regulator. However, the output buffers and solid state switches receive +5 volt power from the control panel. This is done in order to protect other devices which are connected to the data bus. The problem is that most CMOS devices are damaged if a voltage which is even just a little above their power-supply voltage is applied to another of their terminals. Thus, if the 5 volt regulator on the EPROM board supplies a higher voltage than the 5 volt regulator on the control panel, unacceptably high voltages may appear on the data bus. These are applied to the various solid state switches which are connected to it. On the other hand, if the 5 volt regulator on the control panel supplies a higher voltage than the 5 volt regulator on the EPROM board, then unacceptably high voltages may be applied to the solid state switches on the EPROM board. The solution is to use a common supply to power all devices which are connected to the data bus. Of course, if we use an off-board supply to power the switches, we also have to protect the inputs (including the control inputs) of the switches. For this purpose, we use 4049 inverter-buffers or 4050 buffers. These devices can tolerate a higher voltage at an input than is applied to their power terminal.

The +5 volt regulator is used in conventional fashion. However, a 3-terminal, -9 volt regulator is not available from the usual sources. This is not a problem since we can use a -5 volt regulator to provide -9 volt power in the way which is shown.

Construction

The foil side of the PC board and the component layout are shown in Figs. 18-3 and 18-4, respectively. Those who choose one or the other alternatives to PC-board construction shouldn't

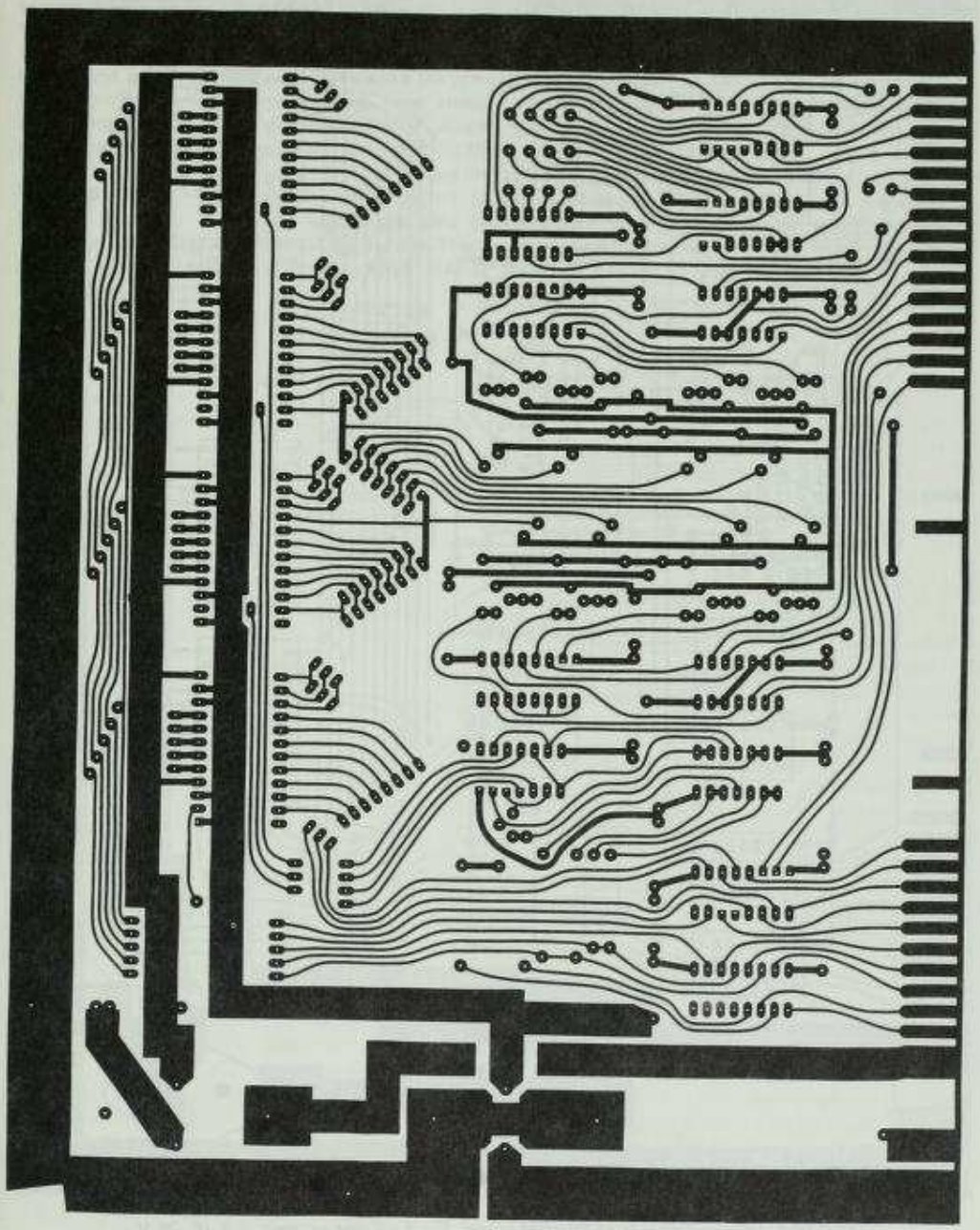


Fig. 18-3. Foil side of EPROM board.

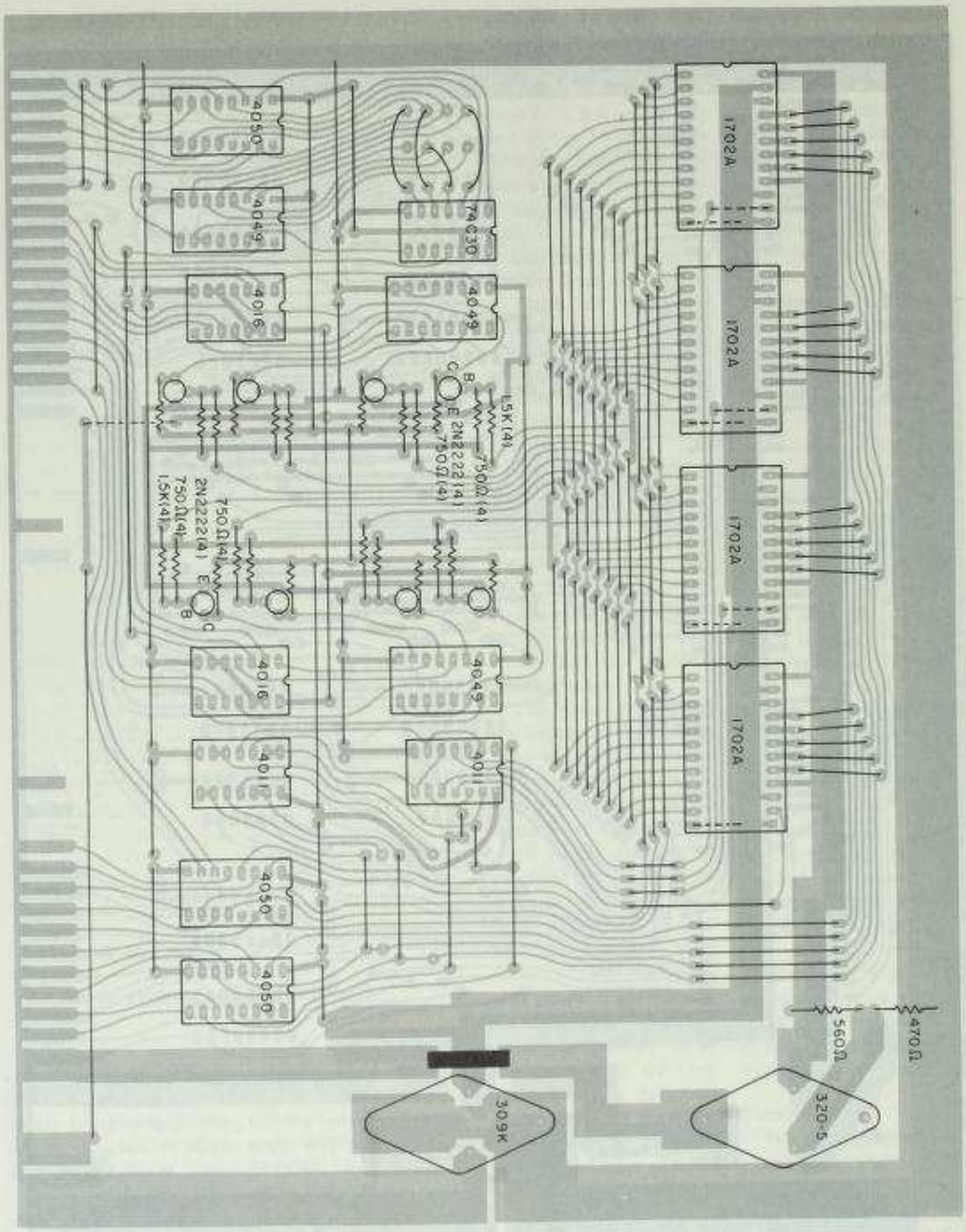


Fig. 18-4. Component side of EPROM board. Tie .1 uF disc ceramic; 6.8 uF, 25 V tantalum; and 47 uF, 50 V electrolytic capacitors across outputs of regulators. Use heatsinks on regulators. Dotted lines indicate jumpers on underside of board.

have any major problems. However, only that part of the circuit on the output side of the 1702As was breadboarded during design and construction of the prototype. Other design work was done directly on PC boards.

Verifying Proper Operation

We can verify that the EPROM works properly in about the same way that we tested the DIPROM.

After verifying that each power terminal at each IC receives the proper voltage, we check that the address which appears on ADDRESS lines A0-A7 is passed on via the 4050 buffers to the appropriate pins of the 1702A sockets. Then, with the 4011 NAND gate in place, we verify that the four possible combinations which can be applied to A8 and A9 generate CHIP-SELECT signals (low-level) at the appropriate 1702A sockets.

We then check that the CONTROL signal to the solid state switches is generated when the desired combination is applied to ADDRESS lines A10-A15.

We check the remaining portion of the circuit by monitoring the data bus (solid state switches on) while we apply high and low levels to each of the on-board DATA lines.

When a programmed 1702A is available, the final step is simply to verify that numbers can be read from it.

Chapter 19
Programming III

Until now, we've been concerned almost entirely with specifics. Indeed, from the very beginning we've had a very specific objective – to build a small system which will perform a few specific, useful tasks. Now that we've achieved our objective, it's time to consider other tasks which our system might perform. We've really only scratched the surface.

As a step in that direction, we'll complete our discussion of the instruction set of the 6502 and see what additional capability that device provides for us. That's the subject of this chapter.

Subroutines

The programs which we've written so far have been quite short as programs go. Such programs are relatively easy to debug. However, as we write longer programs, they're more difficult to deal with. In much the same way as our hardware is broken up into a number of relatively independent modules, so should our long programs be broken up into smaller units, called *subroutines*. We assign single specific tasks to subroutines and then write a program which uses each subroutine in sequence, performing a more complicated task in the process.

Using subroutines produces an even more significant advantage, as well. Once we've written a subroutine, any program we write can use it.

The program which provides the software equivalent of a timer is a good example of what is appropriate for a subroutine. In fact, that program can easily be converted to a subroutine, as shown in Fig. 19-1.

In most respects, the subroutine is the same as the corresponding program which we wrote in Chapter 13. When the timer function is required, the main program executes a jump to the location of the first instruction of the subroutine. That is, it "calls" the subroutine. The subroutine is then executed, producing the same sequence as that which we discussed in the earlier chapter. However, once the subroutine has been executed there's a problem – how can control be returned to the calling program when that program may be in one set of locations at one time and another set of locations at another time?

With what we've discussed so far, there is no way. However, the instruction set of the 6502 includes an instruction which solves the problem. The calling program should not include a JUMP instruction. Rather, a JUMP TO SUBROUTINE (JSR) instruction should be used. When the 6502 encounters that instruction, it notes the address to which to return.

For the final instruction in the subroutine, we use the RETURN FROM SUBROUTINE (RTS) instruction. When the 6502 encounters that instruction, a jump to the return address is executed.

For reasons which involve the internal workings of the 6502, it actually saves the sixteen-bit address of the last word of the JSR instruction rather than the address of the instruction which follows the JSR instruction. This doesn't cause a problem, though, because that address is incre-

LOCATION	INSTRUCTION	CONTENTS
FFC0		
	as in Fig. 13-7.	
FFDA		
FFDB	RTS	60

Fig. 19-1. Converting the timer program to a subroutine.

mented by 1 when the RTS instruction is executed.

The 6502 saves the return address by storing it in the RAM. Of course, since we use the RAM too, we have to have some way to be sure that our programming requirements and the requirements of the 6502 don't conflict. The key to this is a register within the 6502 — the STACK POINTER REGISTER (SP). We'll see where the name comes from in a moment. For now, we'll simply describe it as an eight-bit register which holds the less-significant bits of the address where the 6502 stores (among other things) return addresses. We can dictate what that address is, because we can specify the contents of the register by loading the XR with the desired (eight-bit) address and executing a TRANSFER XR TO SP (TXS) instruction.

When the 6502 stores a number, it places the contents of the pointer register on the eight less-significant lines of the address bus, and places 01_{16} on the eight more-significant lines of the address bus. If the RAM contains only 256 locations, as is the case with ours, the 01 is of no significance, and the 6502 uses a location in page zero. However, if the RAM is larger, the 6502 uses page one, leaving all of page zero to us.

Each time the 6502 writes a number into the RAM, it decrements the SP. If we initialize the SP to 255, the 6502 will start at the top of page zero (or one) and work its way down as it stores numbers. In effect, it forms a stack of numbers — hence, the name of the SP.

At this point we should clearly understand that of the instructions which we've discussed, only the JSR instruction causes the 6502 to write numbers into the RAM under control of the SP. With that, we've solved one of the problems which accompany the use of subroutines — we've found a way to save a return address. However, if we're using a subroutine such as the timer program, there may be a second problem to contend with.

As long as the same delay is always required there is no problem. However, if different delays are required at different times, then we must have a way of changing the number of times one or the other of the loops in the timer program is traversed. The general problem is that we need to be able to pass information to a subroutine. Then too, there are times when it's useful to be able to pass information, such as the result of a calculation, back from a subroutine. One way to do this is to use the stack. We can store the contents of the ACR on the stack by means of the PUSH ACR (PHA) instruction. If for one reason or another we want to save the contents of the SR, we can do this by using the PUSH PROCESSOR STATUS (PHP) instruction. We can load the ACR and the status register from the stack by using the PULL ACR (PLA) and PULL PROCESSOR STATUS (PLP) instruction, respectively.

If we need to examine the contents of the SP, we use the TRANSFER SP TO XR (TSX) instruction. Once the transfer has taken place, we examine the XR.

Interrupts

The final set of instructions which the 6502 recognizes is made up of those which involve interrupts.

In Chapter 9 we saw that the sequential execution of a program can be interrupted by application of an appropriate signal to one or the other INTERRUPT pins on the 6502. We saw that one of these interrupts is unconditional. That is, there is no way (short of electrically disconnecting the lead from the pin) to prevent the 6502 from responding to the signal. This is the non-maskable interrupt. On the other hand, the remaining interrupt can be disabled by software. The appropriate instruction is the SET INTERRUPT (SEI) instruction. This instruction sets a bit to 1 in the SR, indicating that signals on the INTERRUPT REQUEST lines are to be ignored. The bit can be reset to 0 by the CLEAR INTERRUPT (CLI) instruction.

When an interrupt is signaled, the 6502 completes the instruction in progress and stores on the stack the contents of the SR and the address of the next sequential instruction. It then fetches the contents of locations FFFA and FFFB, or FFFE and FFFF, depending on which interrupt was signaled. The contents are taken to be the address of the next instruction.

After the program which starts at that location is completed, we return to the program which was interrupted by using the RETURN FROM INTERRUPT (RTI) instruction. The 6502 pulls from the stack the return address and the contents of the SR which it stored there, restores the SR, and fetches the instruction which is stored at the return address.

When we're debugging a program, we can determine the effect of an interrupt by including the BREAK (BRK) instruction in the program. When the 6502 excounters this instruction, it performs the same sequence of operations as it does in response to a signal on the interrupt request line.

In the last chapter, we encountered an instance where using an interrupt would be convenient. If we tie the key pressed line from the keyboard to one of the interrupt lines, the 6502 will not have to continually check the status of the line. Rather, it can go about other tasks and accept a number from the key-

board in response to an interrupt.

At this point, we've discussed all the instructions which the 6502 recognizes. We'll conclude this final chapter on programming by discussing the remaining modes of addressing which are available with the 6502.

Indirect Addressing

With the possible exception of immediate addressing, we can lump together the various methods of addressing which we've discussed into the category of direct addressing. As the name implies, an instruction which involves direct addressing contains the actual address of the operand, or at least a base address which is modified by the contents of the XR or YR.

The 6502 provides other methods of addressing which we'll lump together under the category of indirect addressing. An instruction which involves indirect addressing contains not the address of the operand, but the address of a location which in turn contains the address of the operand.

While that may seem like an indirect way to go about our business, it's really very useful. For example, we may write a subroutine which performs a complex calculation on a series of numbers that are stored consecutively somewhere in memory. If the numbers are always stored in the same locations, there's no problem. However, if they're not, then we may have to change the addresses (direct) which are contained in many instructions. If the program is contained in the RAM, that's practical, but hardly convenient. If the program is contained in the EPROM, it's not worth considering. The simple solution is to use indirect addressing in any instance where a problem may arise. In that way, the contents of only one location need be changed — the location which contains the address of the first of the consecutively-stored numbers. The design of the 6502 requires that the location which contains the address must itself be located in page zero. Since the RAM is located there, changing the contents of the location is no problem.

The 6502 provides three variations of indirect addressing. These are summarized in Fig. 19-2, along with zero-page (direct) addressing for comparison.

When zero-page addressing is used, the instruction contains two numbers — the op code and the eight less-significant bits (the low order byte, ADL) of the address of the operand. The 6502 applies 00_{16} ADL to the address bus and the desired data appears on the data bus, in response.

In indirect addressing, the instruction also contains two numbers and the first is again the op code. In this case though, the second is the low-order byte of the indirect address (IAL) — the address of the address of the data. The 6502 applies 00 IAL to the address bus and the low-order byte of the address (ADL) of the data appears on the data bus. The 6502 stores this and then applies 00 IAL+1 to the address bus. In response, the high-order byte of the address (ADH) of the data appears on the data bus. The 6502 then applies ADH ADL (the direct address of the data) to the address bus and the data appears on the data bus, in response.

Indexed-indirect and indirect-indexed addressing are similar, except that either the XR or YR is used as shown.

Examples of how instructions are coded for assembly language are shown in the figure.

For the 6502 only the JMP instruction can use ordinary indirect addressing. Several instructions can use indexed-indirect or indirect-indexed, as shown in Appendex I. Of course if the XR or YR, as appropriate, is set to zero, then either indexed method is reduced to ordinary indirect addressing.

	METHOD OF ADDRESSING			
	Z-PAGE (DIRECT)	INDIRECT	INDEXED INDIRECT	INDIRECT INDEXED
INSTRUCTION	OP CODE, ADL	OP CODE, IAL	OP CODE, IAL	OP CODE, IAL
APPLIED TO ADDRESS BUS	00, ADL	00, IAL	00, IAL+X	00, IAL
		00, IAL+1	00, IAL+1+X	00, IAL+1
RESULT ON DATA BUS	DATA	ADL	ADL	ADL
		ADH	ADH	ADH
APPLIED TO ADDRESS BUS	— —	ADH, ADL	ADL, ADH	ADL+Y, ADH+Y
RESULT ON DATA BUS	— —	DATA	DATA	DATA
NOTATION	LDA NUMB	*	LDA (NUMB,X)	LDA (NUMB),Y

*See text

Fig. 19-2. Summary of indirect addressing.

Chapter 20
Advanced Applications

Tidying The Existing System

If we use either the digital signal generator or the digital voltmeter to any extent, we'll soon be aware of some limitations. We didn't dwell on the limitations as we were building the digital portion of the system. However, now that that's done, we might profitably spend some time improving the analog portion.

A number of improvements could be made to the digital signal generator. First, the signal from the current-to-voltage converter could be sent directly to a small accessory box which could also contain the low pass filter and the buffer. The gain and offset of the buffer could be made adjustable, and the components in the filter could be made switch-selectable. Finally, the program by which the generator is implemented could be re-written so that the numbers which form the image of the waveform are transferred into the RAM before use. In that way, zero-page addressing could be used in the generator program itself, and the loop in the program would require less time per traverse. The change would increase the maximum frequency which the generator could provide.

Just as the versatility of the signal generator could be increased, a number of additions could be made to the digital voltmeter. First, switchable-selectable voltage dividers could be placed ahead of the input buffer in order to provide other ranges (0-0.255, 0-25.5 and 0-255 volts full-scale, for example). A rectifier would add the capability to measure ac voltage, as well.

If a current-to-voltage converter were added, currents could be measured. Then too, the function of an ohmmeter could be added by driving the summing junction of an additional op amp with a fixed (but switch-selectable) current source. If the unknown resistance were then placed in the feedback loop of the op amp, the voltage output of that op amp would be directly proportional to the magnitude of the unknown resistance (see Chapter 3).

If proper scaling factors were chosen, little additional scaling of the resultant equivalent numbers would be necessary in any of these additions.

All the analog functions of the system could be housed in one enclosure, producing a handy test instrument. A small control terminal could be assembled by collecting together the keyboard, the digital display and the switches from the control panel in a single enclosure. A set of LED indicators which were driven by an output port could be added in order to signal such things as "system waiting for input from keyboard," and the like.

Additional Hardware

Once the expanded version of the system were tidied up, other pieces of hardware could be added to make a good, solid, small-computer system.

The first addition

probably should be a pair of cassette interfaces. Even one would be useful, but to write long programs in assembly language requires a pair, since the system may not be able to hold all the information which is required during the course of an assembly. Typically, the rough draft of the input to the assembler program is written onto a tape. That tape, in turn, provides the input to an editor program, which is used to correct errors. The corrected output (from the editor program) is then used as the input to the assembler program. The assembler program, in turn, writes the assembled program onto yet another tape for storage.

If we write programs which are long enough to require the hardware which we just discussed, we'll need a fairly large RAM to hold the programs as each is used. As much as 4096 words of RAM would be a useful addition to the system. However, if we attempt to lay out a single-sided PC board for such a memory, we'll find that the board is unacceptably large. A better approach would be to buy one or the other of the "4k naked RAMs" which are commercially available. These are built on double-sided boards which are acceptably small. A "mother" board would be necessary in order to provide power for the ICs and buffering on the DATA and ADDRESS lines, but such a board would be fairly easily built.

When the editor program was mentioned a few sentences ago, we glossed over a rather important point — we require some way to see what's been written on a tape if we're to find and correct errors. For this purpose a video display would be ideal. Such a display consists in part of a video picture tube and the necessary scan circuits. Typically, the display also contains one or two thousand words of RAM which contain (in ASCII) the information that is to be displayed. The RAM is accessable not just to the display, but to the system as well, so that the contents can be updated as required.

The final piece of hardware which we'll mention at this point would be useful in some of the applications that we'll discuss later in the chapter. The applications require an output device which will produce written "hard copy." At the present time (early 1977), used Teletypes® and electric typewriters are probably the better bets. However, the market place is rapidly changing so that this soon may not be true. The ads in the hobby magazines should be consulted for up-to-date information.

Additional Software

If this book had been written a year earlier, this section would have been quite long. However, during the past few months, enough software has become available for the 6502 that we needn't dwell on the topic.

There is, however, a short program which we can write that would be quite useful. The program would accept a four-digit hexadecimal number from the keyboard as input, would use that number as an address, fetch the contents of the location, and display in the digital display both the address and the contents. It would be very helpful in determining that locations contained the proper numbers.

Advanced Applications

In the remainder of the book, we'll consider four additional applications for our system. The first involves using the system as a controller for a digital-calculator chip.

The block diagram of a hypothetical hand-held calculator is shown in Fig. 20-1. The calculator performs only simple arithmetic operations on decimal digits. Major components include the keyboard, the display, and the calculator chip.

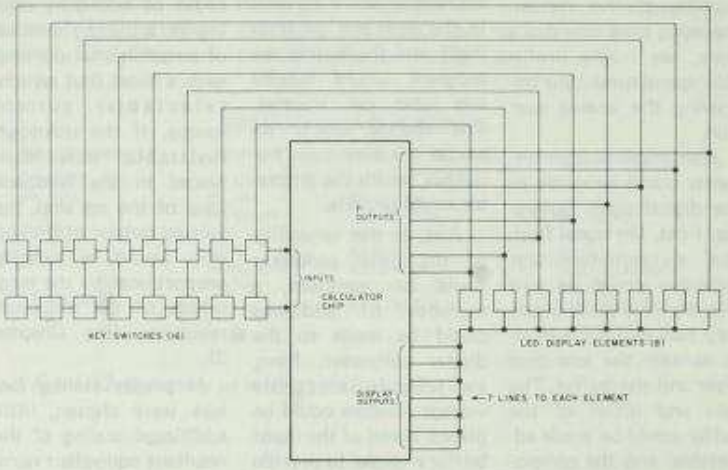


Fig. 20-1. A simple hand calculator.

While it's the chip that interests us most, let's briefly examine how the calculator works.

A series of pulses is transmitted from each of the P terminals. At any given time, one and only one of the P lines is high. When a key is pressed, the high level is passed on to one of the input terminals. Since the calculator knows which P line is high and to which input the high level is applied, it knows which key is pressed.

The pulsed outputs from the P terminals serve another purpose as well. When a pulse appears, it pulls high the common anode of one of the LED display elements, in effect selecting that element. At the same time, the calculator applies logic levels to the output lines to display a character. If a line is pulled low, the corresponding segment of the selected element is lit. If a line is pulled high, the corresponding segment remains unlit. The process is then repeated for the next element, then the next, and so on, without end. Because the "scan" rate is high (a few hundred kHz or so), flicker is absent.

To interface the calculator to our system, we must do two things. The first is to simulate electronically what happens when a key is pressed. This can be done by means of the circuit which is shown in Fig. 20-2.

A two-input NAND gate is used to simulate a key. One input to a gate is the appropriate P signal. The other input is a bit from an output port of the system.

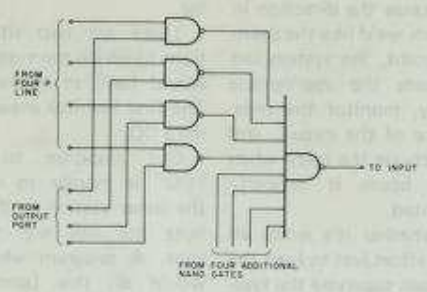


Fig. 20-2. Circuit to simulate keypressing.

Initially, 0s are written into each bit of the output port. Thus, regardless of the state of any P line, the outputs of all the two-input gates are high. This means that the output of the eight-input NAND gate is low. To the calculator, this means that no key is pressed.

We "press a key" by writing a 1 into the corresponding bit of the output port. When the calculator pulses high the corresponding P line, the output of the two-input gate goes low, and that of the eight-input gate goes high. To the calculator, this means that a key is pressed. As long as the contents of the output port remain the same, the "key remains pressed." We "release the key" by writing a 0 back into the bit.

We not only must put information into the calculator, we must get it back out as well. A brute-force approach would be to tie the inputs of 8 eight-bit latches to the output lines and use the pulses on individual P lines to trigger individual latches. The system could then read the contents of each latch sequentially and store the results in eight consecutive locations in memory.

A more practical approach would be to use a single latch. The pulses from each individual P line could then be gated to the latch so that the eight results would be stored sequentially in the latch. The system would read each result as it was stored, until all eight were read.

To use the interfaced chip, we'd write a program which first would "press the appropriate keys" in the desired sequence. A delay would then be produced during which the calculator chip would perform the calculation. The delay could be fixed at a duration that is known to allow sufficient time for even the most time-consuming calculation. Alternatively, the system could continuously check for the appearance of a decimal point, since one would appear in any result.

Once the system read the results of a particular calculation, they would exist as a series of eight numbers in the memory of the system. The numbers would not be directly usable, since they would have been encoded by the calculator chip into a form which is appropriate for driving the LED display elements. For this reason, we would have to write a program which would decode the representations into their binary equivalents. Alternatively, a decoder circuit could be inserted between the output of the calculator chip and the input of the latch.

The example which we just discussed involved a simple, hypothetical chip. This was done in order to simplify the discussion. However, there's no reason why a more complicated chip couldn't actually be used. In fact, we could interface even a programmable calculator to the system. Whether we go that far or not, we could provide a way to make some fairly complicated calculations which are involved in the beam antenna pointer that we'll consider now.

If we interface an antenna rotator to our system, we'll be able to let the system take over the task of pointing our beam. By writing the appropriate program we could type, say, ZL on the keyboard of the system, and the system would point the beam at New Zealand. If we examine how an antenna rotator works, we can see what sort of interface is

needed.

The control switch of a typical antenna rotator is usually a DPDT, center-off switch. The drive motor rotates in one direction or the other if one set of contacts or the other is closed. The direction of rotation depends on which set is closed. The switching functions could be automated by replacing the manually-operated switch with system-driven relays. The relays could be interfaced to the system as we discussed at the end of Chapter 13.

The direction indicator of a typical antenna rotator consists, in part, of a potentiometer which is mechanically linked to the shaft of the rotator. If the wiper and one end of the potentiometer winding were connected to the digital ohmmeter which we discussed earlier, then our system could determine in which direction the beam is pointing by measuring the resistance of the circuit.

If we load into the system a number which indicates the direction in which we'd like the beam to point, the system can actuate the appropriate relay, monitor the resistance of the circuit, and deactuate the relay when the beam is properly pointed.

Whether it's worth all the effort just to have the system supervise the typical beam is open to question. However, such a system would be very useful for tracking OSCAR or a weather satellite. If a calculator chip were also interfaced to the system, the system could calculate orbits and know just what sequence of bearings to use during a pass. If a clock chip were also interfaced to the system, the system could call us in time to prepare for the pass.

As we use our system, we'll find that one of its major virtues is that it can perform routine chores, freeing us to do other things. Log keeping for an amateur radio station isn't the most difficult chore, but if we program our system to do it, a little more effort would produce a contest monitor.

There are two situations to which the system would have to respond. The first involves answering a CQ.

Our response to a "CQ" is merely to call the other station, and to note the call and the time. A program which would do this (among other things) wouldn't be difficult to write. It would accept the other call and a number, N, as an input. Our own call would have been included in the program already. After typing in the call and the number, we'd press, say, the RETURN key to indicate to the system that it had control. The program would then construct the memory image of the other station's call, and append the image of "DE" and our own call to it. The system would then send the combination N times, append K, and stop sending. It would then note the time and print the call and time in the log. Assuming that the other station responded, we would type his report to us and the system would add that to the log. In turn, we would type our report to him and, the system would send that, our QTH, and our name, on command.

As the QSO progressed, the system would send the necessary exchange on command. When the QSO was finished, the system would send the final exchange and note the time.

We'll not examine the sequence which is involved if we send the "CQ" since it's so similar.

The thing to recognize is that none of the programming involved is unfamiliar to us. Much of the machine code from the CW typewriter program could be used intact, for example.

To expand the program to serve as a contest monitor would require that the system store the calls of all stations as we worked them and that it search the list before transmitting a response. Such a program sequentially fetches and compares numbers, and wouldn't be difficult to write.

Appendices

Appendix I — Instruction

INSTRUCTION SET FOR 6502

	ADC	AND	ASL	BCC	BCS	BEQ	BIT	BMI	BNE	BPL	BRK	BVC	BVS	CLC	CLD
IMMEDIATE	69	29													
ABSOLUTE	6D	2D	0E				2C								
ZERO PAGE	65	25	06				24								
ACCUMULATOR			0A												
IMPLIED											00			18	D8
(INDIRECT,X)	61	21													
(INDIRECT),Y	71	31													
ZERO PAGE,X	75	35	16												
ABSOLUTE,X	7D	3D	1E												
ABSOLUTE,Y	79	39													
RELATIVE				90	B0	F0		30	D0	10		50	70		
INDIRECT															
ZERO PAGE,Y															
FLAGS AFFECTED: N	X	X	X				M7								
Z	X	X	X				X								
C	X		X											0	
I															
D															0
V	X						M6								

INSTRUCTION SET FOR 6502 (CONT'D)

	CLI	CLV	CMP	CPX	CPY	DEC	DEX	DEY	EOR	INC	INX	INY	JMP	JSR	LDA
IMMEDIATE			C9	E0	C0				49						A9
ABSOLUTE			CD	EC	CC	CE			4D	EE			4C	20	AD
ZERO PAGE			C5	E4	C4	C6			45	E6					A5
ACCUMULATOR															
IMPLIED	58	B8					CA	88			E8	C8			
(INDIRECT,X)			C1						41						A1
(INDIRECT),Y			D1						51						B1
ZERO PAGE,X			D5			D6			55	F6					B5
ABSOLUTE,X			DD			DE			5D	FE					BD
ABSOLUTE,Y			D9						59						B9
RELATIVE															
INDIRECT													6C		
ZERO PAGE,Y															
FLAGS AFFECTED: N			X	X	X	X	X	X	X	X	X	X			X
Z			X	X	X	X	X	X	X	X	X	X			X
C			X	X	X										
I	0														
D															
V		0													

INSTRUCTION SET FOR 6502 (CONT'D)

	LDX	LDY	LSR	NOP	ORA	PHA	PHP	PLA	PLP	ROL	ROR	RTI	RTS	SBC	SEC
IMMEDIATE	AZ	A0			09									E9	
ABSOLUTE	AE	AC	4E		0D					2E	6E			ED	
ZERO PAGE	A6	A4	46		05					26	66			E5	
ACCUMULATOR			4A							2A	6A				
IMPLIED				EA		48	08	68	28			40	60		38
(INDIRECT,X)					01									E1	
(INDIRECT),Y					11									F1	
ZERO PAGE,X		B4	56		15					36	76			F5	
ABSOLUTE,X		BC	5E		1D					3E	7E			FD	
ABSOLUTE,Y	BE				19									F9	
RELATIVE															
INDIRECT															
ZERO PAGE,Y	B6														
FLAGS AFFECTED: N	X	X	0		X			X		X	X			X	
Z	X	X	X		X			X		X	X			X	
C			X							X	X			X	1
I															
D														X	
V															

INSTRUCTION SET FOR 6502 (CONT'D)

	SED	SEI	STA	STX	STY	TAX	TAY	TSX	TXA	TXS	TYA
IMMEDIATE											
ABSOLUTE			8D	8E	8C						
ZERO PAGE			85	86	84						
ACCUMULATOR											
IMPLIED	F8	78				AA	A8	BA	8A	9A	98
(INDIRECT,X)			81								
(INDIRECT),Y			91								
ZERO PAGE,X			95		94						
ABSOLUTE,X			9D								
ABSOLUTE,Y			99								
RELATIVE											
INDIRECT											
ZERO PAGE,Y				96							
FLAGS AFFECTED: N						X	X	X	X		X
Z						X	X	X	X		X
C											
I		X									
D	X										
V											

Appendix II
Sources for Hard-to-Find Materials

In any project of this sort, there are always a few items for which locating a source is difficult. A few are listed below (as possibilities, not necessarily recommendations!).

6502 Microprocessor
MOS Technology, Inc.
950 Rittenhouse Road
Norristown PA 19401

Gold plating kit for
PC board "fingers"
Rapid Electroplating Process, Inc.
1414 S. Wabash Avenue
Chicago IL 60605

Digiclips®
Components Corporation
106 Main Street
Denville NJ 07834

The remaining small components and an EPROM programming service are available from many sources. The best advice is to shop around.

Sam Creason
2940 Arlington Ave.
Fullerton, CA 92635

Appendix III
Using Other Microprocessors

General Considerations

Regardless which of the more popular uPs is involved, its functions can be described in terms of the three buses which are involved in our system: the data bus, the address bus, and the control bus.

Data Bus. Tying the data lines from the uP to the corresponding lines of the bus should produce satisfactory results.

Address Bus. Most popular uPs provide sixteen address lines. In such a case, tying the address lines from the uP to the corresponding lines of the bus should produce satisfactory results. However, this is not always so, as we'll see later.

In some cases, there are solid-state switches within the uP so that the address lines can be disconnected from the address bus on command from an external device. In such a case, there is always an input on the uP to which the command is applied. Tying that pin to the appropriate logic level will permanently enable the address lines.

Some uPs provide sixteen-bit addresses, but dedicate fewer than sixteen lines to addresses. In such cases, the remaining bits typically are available on the data bus early in each instruction cycle. In such a case, the uP provides an output which is used to enable an external latch at the proper time to capture the bits.

Some uPs provide fewer than sixteen address bits. In such cases, the two higher-level lines from the uP should be tied to lines A15 and A14 of the address bus, as a means of selecting ROM, RAM, or I/O. The remaining address lines from the uP should be connected to the corresponding lines of the bus.

Control Bus. Most of the thought devoted to adapting a different uP to the system will involve the control bus. Five signals require particular attention:

R/W
Ø2
$\overline{\text{RESET}}$
$\overline{\text{READY}}$
Ø1

In our system the R/W signal is used to enable a number from an external device onto the data bus. The signal goes high when the uP expects to find a number on the data bus and remains high until the uP has read the number. Most popular uPs provide an output which can be used directly (or in complemented form) for this purpose.

In our system, the Ø2 signal is used to enable the latch function of an external device when the uP places a number onto the data bus. The signal goes high after the number is on the bus, and then goes low before the number is removed. Most popular uPs provide an output which can be used directly (or in complemented form) for this purpose.

Most popular uPs provide a RESET input which can be properly activated by using the $\overline{\text{RESET}}$ signal (or its complement) from our system.

In our system, the $\overline{\text{READY}}$ line provides a

means to stop the operation of the uP without removing power. Most popular uPs provide an input which is directly compatible with the READY signal or its complement.

In our system, the Ø1 and READY signals together provide a capability for single-cycle operation. Generalizing about other uPs is difficult because the details of timing vary from one uP to the other. However, whatever signal is chosen to simulate Ø1 must be available even when the operation is performed which is equivalent to pulling the READY line low.

Finally, most uPs have some sort of interrupt which resemble NMI and IRQ in their function.

Using A Z80 Microprocessor

Of all the uPs which are available, the Z80 seems the most likely alternative to the 6502. Among other reasons, any program which will run on the 8080 will run on the Z80[1]. This means that a lot of inexpensive software is readily available.

The functions of most of the inputs and outputs of the Z80 are described in Fig. AIII-1. Only those that are used in a minimal implementation are included. The hardware manual[2] contains a complete description of the uP.

A minimal implementation of the Z80 in our system is shown in Fig. AIII-2. Let me emphasize that while this circuit almost certainly will work, it has not been breadboarded nor tested in the prototype of our system.

An MC 4024 is used as a clock generator since there is none on-board the Z80. BUSRQ is pulled up to +5 volts via a 10k resistor, permanently enabling the address lines. The complements of RD and WR are used to simulate R/W and Ø2. The data lines are tied to the data bus.

The complements of address lines A14 and A15 are tied to the address bus, in order to accommodate the requirements of the Z80. When the RESET input is taken low, the Z80 looks in location 0000 (rather than FFFC) for the first byte of information. Further, location 0066 is used in response to a non-maskable interrupt and (under some circumstances) location 0038 is used in the response to a maskable interrupt. Thus, the DIPROM should be located toward the low end of memory. Inverting address lines A14 and A15 accomplishes this. The circuit which generages the *read* signal for the DIPROM (74C30 on control panel) must also be modified slightly. The jumpers should be located so that a *read* signal is generated when A14 and A15 are high and A13, A12, A11, A10, A9, and A8 are low.

Inverting address lines A14 and A15 also places the RAM near the high end of memory and modifies the addresses of the I/O ports as well. For example, if an I/O address was 8200, it becomes 4200.

A0-A15	The address lines. There are tri-state outputs (solid-state switches) within the Z80 so that the lines can be disconnected internally from the address bus.
BUSRQ	An input, active low, by which an external device can request use of the address bus. In a minimal system, this pin is tied to +5 volts.
D0-D7	The data lines. They function in the same way as the data lines of the 6502.
RD	A tristate output, active low, which indicates that the MP expects to find a number on the data bus.
WR	A tristate output, active low, which indicates that the number which appears on the data bus is to be stored in memory or I/O which has been configured as memory.
Ø	An input which is to be driven with a single phase clock generator.
WAIT	An input, active low, similar in function to the READY input of the 6502.
RESET	An input, active low, which performs the same function as the RESET input of the 6502.
INT	An input, active low, similar in function to the IRQ input of the 6502.
NMI	An input, negative-edge triggered, similar in function to the NMI of the 6502.

Fig. AIII-1. Functions of the lines of the Z80.

LINE	PIN	LINE	PIN
A0	30	D0	14
A1	31	D1	15
A2	32	D2	12
A3	33	D3	8
A4	34	D4	7
A5	35	D5	9
A6	36	D6	10
A7	37	D7	13
A8	38		
A9	39		
A10	40		
A11	1		
A12	2		
A13	3		
A14	4		
A15	5		

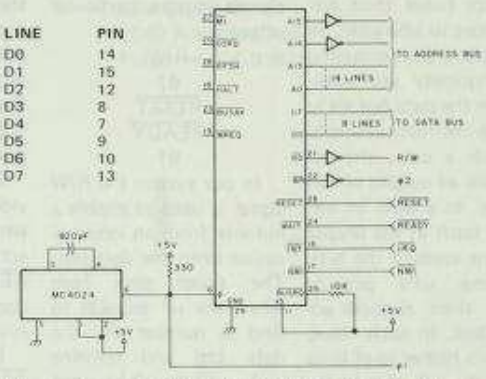


Fig. AIII-2. Proposed minimal implementations using Z80.

[1] Godding, "Is The Z80 The Wave Of The Present?", *Kilobaud*, Jan. 1977, p. 20.

[2] *Zilog Z80 Hardware Manual*, Zilog, 10460 Bubb Rd., Cupertino CA 95014, $7.50.